Mathematics and Computing/Technology
MT365 Graphs, networks and design

MT365 HB

The Open University

MT365
Handbook

2.1

SUP 00518 2

Contents

Introduction

Section 2 Combinatorics

1 **Combinatorics** is concerned with the arrangement, classification and enumeration of finite sets of objects.

Combinatorial problems can be described under one or more of the following interrelated headings.

Existence problems:
does there exist ...? is it possible to ...?

Construction problems:
if ... exists, how can we construct it?

Enumeration problems:
how many ... are there? (counting problem)
can we list them all? (listing problem)

Optimization problems:
if there are several ..., which is the best?

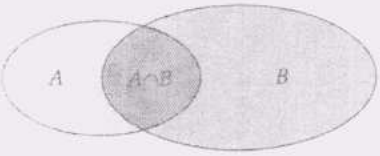
2 Rules of counting

Addition rule: the number of objects in a set can be counted by splitting the set into disjoint subsets and adding together the numbers of objects in each subset. (Subsets are **disjoint** if they have no element in common.)

Multiplication rule: if a counting problem can be split into stages, each consisting of a number of options, then the total number of possibilities is obtained by multiplying the numbers of available options at each stage.

Principle of inclusion–exclusion: if a set S can be split into two subsets A and B, not necessarily disjoint, then the number of elements in S is equal to

(no. of elements in A) + (no. of elements in B)
− (no. of elements common to both A and B).



The set of elements common to both A and B is called the **intersection** of A and B, and is denoted by $A \cap B$.

3 A **binary word** is a string of 0s and 1s; a single *binary digit* (0 or 1) is called a **bit**.

4 An **alkane** is a molecule with formula C_nH_{2n+2}: each carbon atom (C) is bonded to exactly four atoms, and each hydrogen atom (H) is bonded to exactly one atom (a carbon atom). If an alkane has four or more carbon atoms, then there exist different alkanes (**isomers**) with the same formula.

5 An $n \times n$ **latin square** is a square arrangement of n letters such that each letter appears exactly once in each row and exactly once in each column.

Section 3 Representing situations diagrammatically

1 A **graph** is a diagram consisting of points, called **vertices**, joined together by lines, called **edges**; each edge joins exactly two vertices.

A **bipartite graph** is a graph in which the vertices split naturally into two sets, with each edge joining a vertex in one set to a vertex in the other.

In a **weighted graph**, each edge has a number associated with it called a **weight** (or sometimes a **cost**). A weight is a *fixed* number.

2 A **digraph** is a diagram consisting of points, called **vertices**, joined together by directed lines, called **arcs**; each arc joins exactly two vertices.

In a **weighted digraph**, each arc has a number associated with it called a **weight** (or sometimes a **cost**).

3 A **network** is a graph or digraph which carries some numerical information. This information may consist of **weights** associated with the edges or arcs, and possibly with some of the vertices.

Section 4 Algorithms

1 An **algorithm** is a systematic step-by-step procedure for solving a problem. It consists of:

- a description of appropriate input data;
- a finite, ordered list of instructions;
- a stop instruction;
- a description of appropriate output data.

A **heuristic algorithm** is an algorithm which does not necessarily give a correct answer to a problem, but usually gives a good approximation to one.

In a **greedy algorithm**, the 'greediest', or best, choice is made at each step, regardless of the subsequent effect of that choice.

The **exhaustion algorithm** examines all possible cases, and then selects the best.

2 **Minimum connector problem**

Given a weighted graph, find a *minimum connector* — a set of edges of minimum total weight connecting some specified vertices.

Kruskal's greedy algorithm

```
START with a finite set of vertices, where each pair is
joined by a weighted edge.
STEP 1   List the weights in ascending order.
STEP 2   Draw the vertices and weighted edge
         corresponding to the first weight in the list,
         provided that, in doing so, no cycle is formed.
         Delete the weight from the list.
Repeat Step 2 until all the vertices are connected, then
STOP.
```

The weighted graph obtained is a minimum connector, and the sum of the weights on its edges is the total weight of the minimum connector.

Prim's greedy algorithm

```
START with a finite set of vertices, where each pair is
joined by a weighted edge.
STEP 1   Choose and draw any vertex.
STEP 2   Find the edge of least weight joining a drawn
         vertex to one not currently drawn. Draw this
         weighted edge and the corresponding new
         vertex.
Repeat Step 2 until all the vertices are connected, then
STOP.
```

The weighted graph obtained is a minimum connector, and the sum of the weights on its edges is the total weight of the minimum connector.

3 Travelling salesman problem

Given a weighted graph, find a route of minimum total weight which passes through each vertex and returns to the starting point.

Heuristic algorithm for the travelling salesman problem

START with a finite set of vertices, where each pair is joined by a weighted edge.

STEP 1 Choose any vertex and find the edge of least weight emerging from it. Draw the corresponding two vertices and join them by two edges, to form a cycle.

STEP 2 Find the edge e of least weight joining a drawn vertex v to a vertex w *not* currently drawn. Draw the new vertex w on the edge of the cycle immediately after v, moving in a clockwise direction.

Repeat Step 2 until all the vertices appear in the cycle, then assign the appropriate weight to each edge, then STOP.

The weighted cycle obtained is a cycle through all the vertices, and its total weight (given by the sum of the weights on its edges) is at most twice the weight of the required minimum cycle.

4 Efficiency of algorithms

The **efficiency** of an algorithm is a measure of the time it takes to solve a problem.

A **polynomial-time algorithm** is an algorithm for which the time taken is proportional to n^k, for some fixed number $k > 1$, or to some function $f(n)$, where there is some fixed number $k > 1$ such that $f(n) \leq n^k$ for all $n \geq N$, for some fixed number N, where n is a parameter associated with the problem. Such an algorithm is usually efficient. Problems for which polynomial-time algorithms exist are said to be **tractable**. Problems that are not tractable are said to be **intractable**.

An **exponential-time algorithm** is an algorithm for which the time taken is proportional to k^n, for some fixed number $k > 1$, or to some function $f(n)$, where there is some fixed number $k > 1$ such that $f(n) \geq k^n$ for all $n \geq N$, for some fixed number N, where n is a parameter associated with the problem. Such an algorithm is usually efficient *only* for problem instances of small size.

There is an important class of problems called **NP-complete problems**. The types of problem in this class have the following properties:

- no polynomial-time algorithm has yet been found for any of them;
- no one has proved that polynomial-time algorithms do not exist for any of them;
- it has been shown that, if a polynomial-time algorithm could be found for any *one* of them, then polynomial-time algorithms must exist for *all* of them;
- it has been shown that, if it could be proved that no polynomial-time algorithm exists for *one* of them, then polynomial-time algorithms would not exist for *any* of them.

It is currently believed that no polynomial-time algorithm exists for any of the NP-complete problems — that is, that NP-complete problems are intractable.

Section 5 Mathematical modelling

1 The mathematical modelling process

A **model** is an abstraction of a real or imagined system. An important feature is that the important components of the real system should be represented by parts of the model.

The **mathematical modelling process** may involve the following stages.

1 Identify the practical problem.
2 Carefully describe it in ordinary language.
3 Formulate the description as a mathematical problem.
4 Analyse and try to solve the mathematical problem.
5 Interpret the mathematical solution in terms of the original problem.
6 Examine the proposed practical solution to determine whether it is satisfactory.

The mathematical formulation of a practical problem is called a **mathematical model**.

2 Location problem

Given a network with n vertices, representing places to be served by a number of facilities, the **location problem** involves one or both of the following:

(a) finding the *least* number of facilities required so that the maximum distance between each place and its nearest facility does not exceed a given value d;

(b) finding the *best* locations on the network for a given number m of facilities so that the maximum distance between each place and its nearest facility is a minimum.

A **region** of a network is a set of points on the network such that the set of vertices of the network that can be reached by travelling a given distance δ from *any* of the points in the region is always the same. A region may be a single point, part of an edge or parts of several edges.

Algorithm of Christofides and Viola

Given a location problem, START by drawing the network. Carry out the following steps for each distance δ.

STEP 1 Penetrate a distance δ from each vertex along the edges of the network.

STEP 2 Identify the regions created, and label them with the combined labels of all the vertices that are within a distance δ of some point in the region.

STEP 3 Draw the bipartite graph consisting of **region** vertices on the left, representing the regions created in Step 2, and **place** vertices on the right, representing the vertices (places) of the network. Join a region vertex to a place vertex if the place is within a distance δ of some point in the region.

STEP 4 Use the bipartite graph to determine the minimum number and best locations of the facilities that can serve all the places on the network for the distance δ.

In the case of location problems of type (a), carry out Steps 1–4 once, with δ equal to the specified distance d.

In the case of location problems of type (b), begin with $\delta = 0$ and increase δ in stages in such a way that each *new* region created at each stage consists of just a single point. STOP when the minimum number of facilities determined in Step 4 equals the specified number m.

Graphs 1 Graphs and digraphs

Section 1 Graphs

1 A **graph** consists of a set of elements called **vertices**, and a set of elements called **edges**. Each edge is associated with two vertices, and is said to **join** them.

Two or more edges joining the same pair of vertices are **multiple edges**. An edge joining a vertex to itself is a **loop**.

A graph with no loops or multiple edges is a **simple graph**.

The vertices v and w are **adjacent** vertices if they are joined by an edge e. The vertices v and w are **incident** with the edge e, and the edge e is **incident** with the vertices v and w.

Two graphs are the *same* if they have the same vertices and edges.

The **complement** of a simple graph G is obtained by taking the vertices of G and joining two of them whenever they are *not* joined in G.

2 Two graphs G and H are **isomorphic** if H can be obtained by relabelling the vertices of G, that is, if there is a one–one correspondence between the vertices of G and those of H such that the number of edges joining each pair of vertices in G is equal to the number of edges joining the corresponding pair of vertices in H.

3 A **subgraph** of a graph G is a graph all of whose vertices are vertices of G and all of whose edges are edges of G.

4 In a graph, the **degree** of a vertex v is the number of edges incident with v, with each loop counted twice, and is denoted by **deg** v.

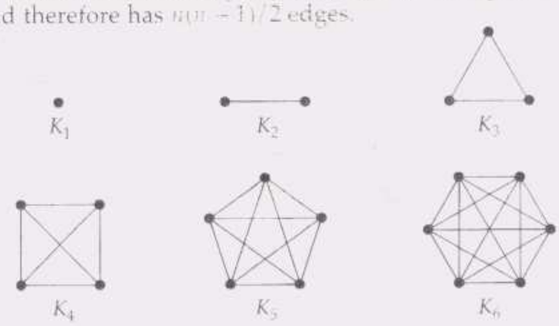
The **degree sequence** of a graph G is the sequence obtained by listing the vertex degrees of G in increasing order, with repeats as necessary.

Theorem 1.1: handshaking lemma
In any graph, the sum of all the vertex degrees is equal to twice the number of edges.

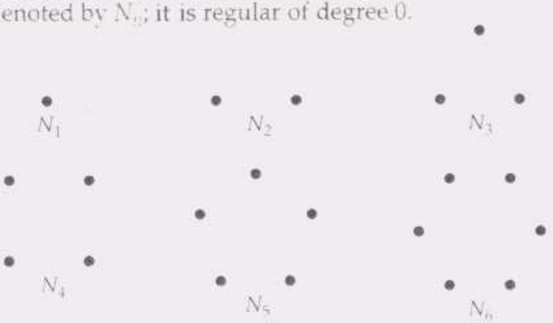
A graph is **regular** if its vertices all have the same degree. A regular graph is **r-regular**, or **regular of degree** r, if the degree of each vertex is r.

Theorem 1.2
Let G be an r-regular graph with n vertices; then G has exactly $nr/2$ edges.

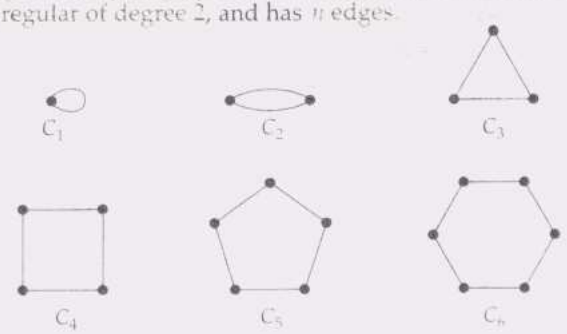
5 In a **complete graph** each vertex is joined to each of the others by exactly one edge. The complete graph with n vertices is denoted by K_n; it is regular of degree $n-1$, and therefore has $n(n-1)/2$ edges.



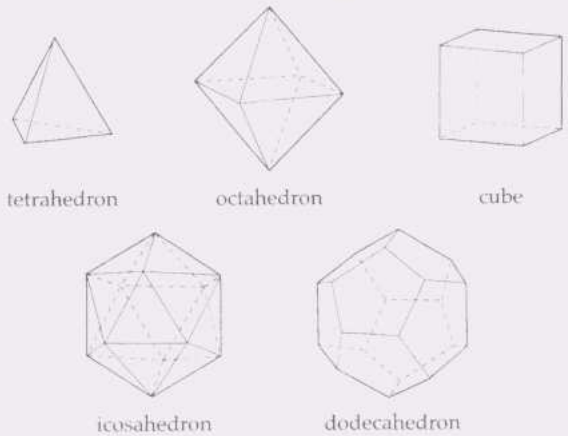
A **null graph** has no edges. The null graph with n vertices is denoted by N_n; it is regular of degree 0.



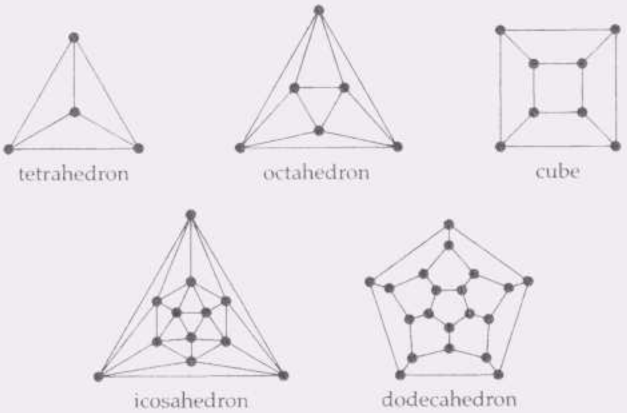
A **cycle graph** consists of a single cycle of vertices and edges. The cycle graph with n vertices is denoted by C_n; it is regular of degree 2, and has n edges.



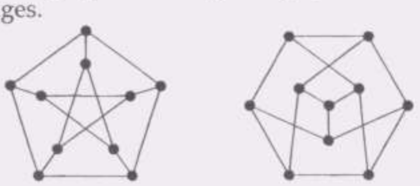
The **Platonic solids** are the five regular solids:



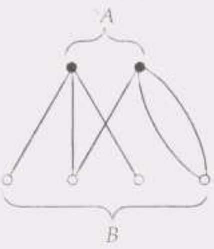
The five **Platonic graphs** are obtained by taking the vertices and edges of each Platonic solid as the vertices and edges of a regular graph.



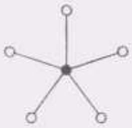
The **Petersen graph** is a 3-regular graph with 10 vertices and 15 edges.



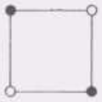
A **bipartite graph** is a graph whose set of vertices can be split into two sets A and B in such a way that each edge joins a vertex in A to a vertex in B.



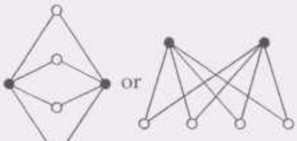
A **complete bipartite graph** is a bipartite graph in which each (black) vertex in A is joined to each (white) vertex in B by just one edge. The complete bipartite graph with r black vertices and s white vertices is denoted by $K_{r,s}$; it has r vertices of degree s and s vertices of degree r, and rs edges.



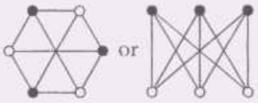
$K_{1,5}$



$K_{2,2}$

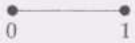


$K_{2,4}$

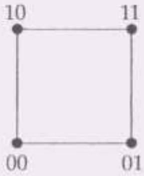


$K_{3,3}$

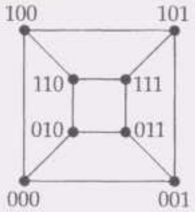
The ***k*-cube (*k*-dimensional cube)** Q_k is the graph constructed by taking as vertices all 2^k sequences of 0s and 1s of length k, and joining two vertices whenever the corresponding sequences differ in just one place; it has 2^k vertices, $k.2^{k-1}$ edges and is regular of degree k.



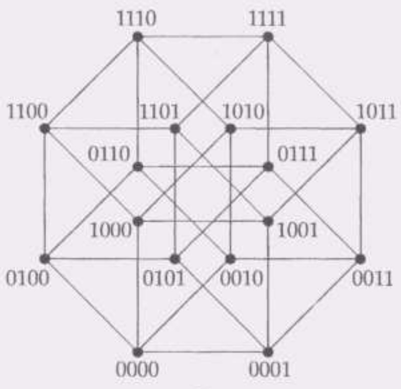
Q_1



Q_2



Q_3



Q_4

6 Walks, trails and paths in a graph

A **walk of length *k*** in a graph is a succession of k edges of the form uv, vw, wx, …, yz. We denote this walk by $uvwx…yz$, and refer to it as a **walk between *u* and *z***.

A **closed walk** in a graph is a succession of edges of the form uv, vw, wx, …, yz, zu.

A **trail** is a walk in which all the edges (but not necessarily all the vertices) are different.

A **closed trail** is a closed walk in which all the edges are different.

An **open trail** is a trail whose ends do not coincide.

A **path** is a trail in which all the vertices are different.

A **cycle** is a closed trail in which all the intermediate vertices are different. A cycle of length 3 is called a **triangle**. Any cycle of a bipartite graph has even length, and any graph in which every cycle has even length is bipartite.

7 A graph is **connected** if there is a path between each pair of vertices, and is **disconnected** otherwise. Every disconnected graph can be split up into a number of connected subgraphs, called **components**.

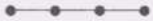
8 A **tree** is a connected graph with no cycles. A tree with n vertices has $n-1$ edges, and there is just one path between each pair of vertices. A tree is bipartite.

A **path graph** is a tree in which there is a path through all its vertices. The path graph with n vertices is denoted by P_n.

P_1

P_2

P_3



P_4

P_5

P_6

Section 2 Eulerian and Hamiltonian graphs

1 Eulerian and semi-Eulerian graphs

A connected graph is **Eulerian** if it contains a closed trail which includes every edge; such a trail is an **Eulerian trail**.

A connected graph is **semi-Eulerian** if it contains an open trail which includes every edge; such a trail is a **semi-Eulerian trail**.

Theorem 2.1
A connected graph is Eulerian if and only if each vertex has even degree.

Theorem 2.2
A connected graph is semi-Eulerian if and only if it has exactly two vertices of odd degree.

Theorem 2.3
An Eulerian graph can be split into cycles, no two of which have an edge in common.

2 Hamiltonian and semi-Hamiltonian graphs

A connected graph is **Hamiltonian** if it contains a cycle that includes every vertex; such a cycle is a **Hamiltonian cycle**.

A connected graph is **semi-Hamiltonian** if there is a path, but not a cycle, which includes every vertex; such a path is a **semi-Hamiltonian path**.

Theorem 2.4: Ore's theorem
Let G be a simple connected graph with n vertices, where $n \geq 3$. If $\deg v + \deg w \geq n$, for each pair of non-adjacent vertices v and w, then G is Hamiltonian.

Section 3 Digraphs

1 A **digraph** consists of a set of elements called **vertices**, and a set of elements called **arcs**. Each arc joins two vertices in a specified direction.

Two or more arcs joining the same pair of vertices in the same direction are **multiple arcs**. An arc joining a vertex to itself is a **loop**.

A digraph with no loops or multiple arcs is a **simple digraph**.

The vertices v and w are **adjacent** vertices if they are joined (in either direction) by an arc e. An arc e that is directed from v to w is **incident from** v and **incident to** w; v is **incident to** e, and w is **incident from** e.

The **underlying graph** of a digraph D is the graph obtained by replacing each arc of D by the corresponding undirected edge.

Two digraphs are the *same* if they have the same vertices and arcs.

2 Two digraphs C and D are **isomorphic** if D can be obtained by relabelling the vertices of C, that is, if there is a one–one correspondence between the vertices of C and those of D such that the arcs joining each pair of vertices in C agree in both number and direction with the arcs joining the corresponding pair of vertices in D.

3 A **subdigraph** of a digraph D is a digraph all of whose vertices are vertices of D and all of whose arcs are arcs of D.

4 In a digraph, the **out-degree** of a vertex v is the number of arcs incident from v, and is denoted by **outdeg** v; and the **in-degree** of v is the number of arcs incident to v, and is denoted by **indeg** v. Each loop contributes 1 to both the in-degree and the out-degree of the corresponding vertex.

The **out-degree sequence** of a digraph D is the sequence obtained by listing the out-degrees of D in increasing order, with repeats as necessary. The **in-degree sequence** of D is defined analogously.

5

Theorem 3.1: handshaking dilemma

In any digraph, the sum of all the out-degrees and the sum of all the in-degrees are each equal to the number of arcs.

6 Walks, trails and paths in a digraph

A **walk of length k** in a digraph is a succession of k arcs of the form uv, vw, wx, ..., yz. We denote this walk by $uvwx \dots yz$, and refer to it as a **walk from u to z**.

A **closed walk** in a digraph is a succession of arcs of the form uv, vw, wx, ..., yz, zu.

A **trail** is a walk in which all the arcs (but not necessarily all the vertices) are different.

A **closed trail** is a closed walk in which all the arcs are different.

A **path** is a trail in which all the vertices are different.

A **cycle** is a closed trail in which all the intermediate vertices are different.

7 A digraph is **connected** if its underlying graph is a connected graph, and is **disconnected** otherwise. It is **strongly connected** if there is a path between each pair of vertices.

8 Eulerian digraphs

A connected digraph is **Eulerian** if it contains a closed trail which includes every arc; such a trail is an **Eulerian trail**.

Theorem 3.2

A connected digraph is Eulerian if and only if, for each vertex, the out-degree equals the in-degree.

Theorem 3.3

An Eulerian digraph can be split into cycles, no two of which have an arc in common.

9 Hamiltonian digraphs

A connected digraph is **Hamiltonian** if it contains a cycle which includes every vertex; such a cycle is a **Hamiltonian cycle**.

Section 4 Matrix representations

1 Adjacency matrix of a graph

Let G be a graph with n vertices labelled 1, 2 ..., n. The **adjacency matrix** $\mathbf{A}(G)$ is the $n \times n$ matrix in which the entry in row i and column j is the number of edges joining the vertices i and j. The matrix is symmetrical about the *main diagonal* (top left to bottom right). If the graph has no loops, then each entry on the main diagonal is 0, and the sum of the entries in any row or column is the degree of the vertex corresponding to that row or column.

2 Adjacency matrix of a digraph

Let D be a digraph with n vertices labelled 1, 2 ..., n. The **adjacency matrix** $\mathbf{A}(D)$ is the $n \times n$ matrix in which the entry in row i and column j is the number of arcs from vertex i to vertex j. The matrix is not necessarily symmetrical about the main diagonal. If the digraph has no loops, then each entry on the main diagonal is 0, and the sum of the entries in any row is the out-degree of the vertex corresponding to that row, and the sum of the entries in any column is the in-degree of the vertex corresponding to that column.

3 Incidence matrix of a graph

Let G be a graph without loops, with n vertices and m edges. The **incidence matrix** $\mathbf{B}(G)$ is the $n \times m$ matrix in which the entry in row i and column j is

1 if the vertex i is incident with the edge j,

0 otherwise.

4 Incidence matrix of a digraph

Let D be a digraph without loops, with n vertices and m arcs. The **incidence matrix** $\mathbf{B}(D)$ is the $n \times m$ matrix in which the entry in row i and column j is

1 if arc j is incident from vertex i,

−1 if arc j is incident to vertex i,

0 otherwise.

Networks 1 Network flows

Section 1 Connectivity

1 Edge connectivity

The **edge connectivity** $\lambda(G)$ of a connected graph G is the *smallest* number of edges whose removal disconnects G.

A **cutset** of a connected graph G is a set S of edges with the properties:

(a) removal of all the edges in S disconnects G;

(b) removal of some (but not all) of the edges in S does not disconnect G.

Two cutsets of a graph need not necessarily have the same number of edges.

The edge connectivity $\lambda(G)$ of a graph G is the size of the *smallest* cutset of G.

A single edge whose removal disconnects a graph is called a **bridge**.

2 Vertex connectivity

The **connectivity** (or **vertex connectivity**) $\kappa(G)$ of a connected graph G (other than a complete graph) is the *smallest* number of vertices whose removal disconnects G.

The **connectivity** $\kappa(K_n)$ of the complete graph K_n ($n \geq 3$) is $n-1$.

A **vertex cutset** of a connected graph G is a set S of vertices (not the whole set of vertices) with the following properties:

(a) removal of all the vertices in S disconnects G;

(b) removal of some (but not all) of the vertices in S does not disconnect G.

Two vertex cutsets of a graph need not necessarily have the same number of vertices.

The connectivity $\kappa(G)$ of a graph G is the size of the *smallest* vertex cutset of G.

A single vertex whose removal disconnects a graph is called a **cut vertex**.

3

Theorem 1.1

For any connected graph G,

$$\kappa(G) \leq \lambda(G) \leq \delta(G),$$

where $\delta(G)$ is the smallest vertex degree in G.

4 *st*-paths in a graph

Let G be a connected graph, and let s and t be vertices of G. A path between s and t is called an ***st*-path**. Two or more st-paths are **edge-disjoint** if they have no edges in common, and **vertex-disjoint** if they have no vertices in common (apart from s and t). Certain *edges* **separate *s* from *t*** if the removal of these edges destroys all paths between s and t. Similarly, certain *vertices* (excluding s and t) **separate *s* from *t*** if the removal of these vertices destroys all paths between s and t.

Theorem 1.2: Menger's theorem (edge form)

Let G be a connected graph, and let s and t be vertices of G. Then the maximum number of edge-disjoint st-paths is equal to the minimum number of edges separating s from t.

If we can find k edge-disjoint st-paths and k edges separating s from t (for the same value of k), then k is the *maximum* number of edge-disjoint st-paths and the *minimum* number of edges separating s from t. These k edges separating s from t necessarily form a cutset. It follows that, when looking for them, we need consider only cutsets whose removal disconnects G into two components, one containing s and the other containing t.

Corollary of Menger's theorem (edge form)

A connected graph G has edge connectivity l if and only if every pair of vertices in G is joined by l or more edge-disjoint paths, and at least one pair of vertices is joined by exactly l edge-disjoint paths.

5 *st*-paths in a digraph

Let D be a connected digraph, and let s and t be vertices of D. A path from s to t is called an ***st*-path**. Two or more st-paths are **arc-disjoint** if they have no arcs in common, and **vertex-disjoint** if they have no vertices in common (apart from s and t). Certain *arcs* **separate *s* from *t*** if the removal of these arcs destroys all paths from s to t. Similarly, certain *vertices* (excluding s and t) **separate *s* from *t*** if the removal of these vertices destroys all paths from s to t.

Theorem 1.3: Menger's theorem (arc form)

Let D be a connected digraph, and let s and t be vertices of D. Then the maximum number of arc-disjoint st-paths is equal to the minimum number of arcs separating s from t.

If we can find k arc-disjoint st-paths and k arcs separating s from t (for the same value of k), then k is the *maximum* number of arc-disjoint st-paths and the *minimum* number of arcs separating s from t.

6

Theorem 1.4: Menger's theorem for graphs (vertex form)

Let G be a connected graph, and let s and t be non-adjacent vertices of G. Then the maximum number of vertex-disjoint st-paths is equal to the minimum number of vertices separating s from t.

If we can find k vertex-disjoint st-paths and k vertices separating s from t (for the same value of k), then k is the *maximum* number of vertex-disjoint st-paths and the *minimum* number of vertices separating s from t. These k vertices separating s from t necessarily form a vertex cutset. It follows that, when looking for them, we need consider only vertex cutsets whose removal disconnects G into two or more components, one containing s and the other containing t.

Corollary of Menger's theorem for graphs (vertex form)

A connected graph G (other than a complete graph) has vertex connectivity k if and only if every non-adjacent pair of vertices in G is joined by k or more vertex-disjoint paths, and at least one non-adjacent pair of vertices is joined by exactly k vertex-disjoint paths.

Theorem 1.5: Menger's theorem for digraphs (vertex form)
Let D be a connected digraph, and let s and t be non-adjacent vertices of D. Then the maximum number of vertex-disjoint st-paths is equal to the minimum number of vertices separating s from t.

8 Let G be a graph with n vertices and m edges. Then G has **optimal connectivity** if $\kappa(G) = 2m/n$.
All graphs with optimal connectivity are regular graphs, but not every regular graph has optimal connectivity.

Section 2 Flows in basic networks

1 A **basic network** is a connected digraph N satisfying the conditions:

(a) N has exactly one **source** and one **sink**;

(b) each arc e of N is assigned a positive number $c(e)$, called the **capacity of** e, which represents the maximum amount of the commodity that can flow through the arc in a given time.
(Capacities are shown in bold type in diagrams.)

2 A **flow** in a basic network N with source S and sink T is an assignment, to each arc e of N, of a non-negative number $f(e)$, called the **flow along the arc** e, satisfying:

(a) the **feasibility condition**: the flow along each arc does not exceed the capacity of the arc, that is, $f(e) \leq c(e)$, for each arc e of N;

(b) the **flow conservation law**: for each vertex V other than S and T, the sum of the flows along the arcs into V is equal to the sum of the flows along the arcs out of V.

Such a flow is sometimes called a *flow from S to T* or an *(S,T)-flow*.

An arc e is **saturated** if $f(e) = c(e)$, and it is **unsaturated** if $f(e) < c(e)$.
(A saturated arc is shown by a thick line in diagrams.)

The total flow out of the source S is always equal to the total flow into the sink T. This common value is called the **value of the flow**, and represents the amount of the commodity flowing through the network.

A **maximum flow** is a flow of largest possible value. (There may be several different flows with the same largest value.)

3 A **flow-augmenting path** in a basic network with source S and sink T is a path from S to T consisting of:

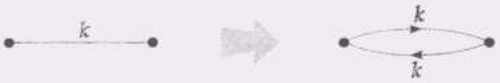
- **forward arcs**: unsaturated arcs directed along the path;
  and, possibly,
- **backward arcs**: arcs directed against the direction of the path and carrying a non-zero flow.

4 Transforming to a basic network

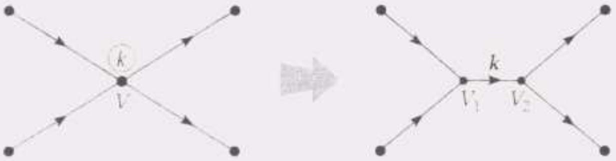
An **undirected network** is one in which flow is allowed in either direction along each edge. A network containing both directed and undirected edges is a **mixed network**.

The transformations to convert a network into a basic network are summarized below, and should be carried out in the order given.

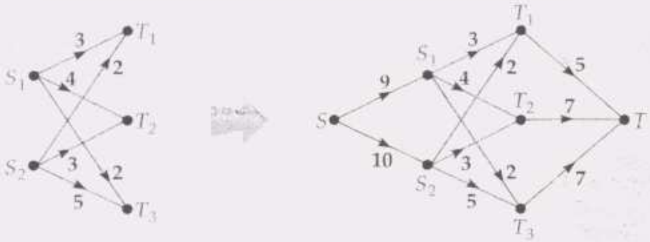
- Replace each undirected edge by two arcs, one in each direction, both with the same capacity as the original undirected edge.



- Replace each vertex V with a capacity restriction k by two vertices V_1 and V_2 joined by an arc from V_1 to V_2 of capacity k. All arcs directed towards V are directed towards V_1, and all arcs directed away from V are directed away from V_2.



- If there are several sources $S_1, S_2, \ldots S_i, \ldots$, join them to a new super-source S. If there are several sinks $T_1, T_2, \ldots, T_i, \ldots$, join them to a new super-sink T. To each new arc SS_i, assign a capacity equal to the sum of the capacities out of S_i; to each new arc T_iT assign a capacity equal to the sum of the capacities into T_i.



General procedure
A variety of network flow problems are solved by the following procedure.

STEP 1 Transform the network into a basic network using the above transformations where necessary.

STEP 2 Solve this basic network problem.

STEP 3 Interpret the solution in terms of the original network.

Section 3 Maximum flows and minimum cuts

1 A **cut** in a basic network with source S and sink T is a set of arcs whose removal separates the network into two disjoint parts X (containing S) and Y (containing T).
The **capacity of a cut** is the sum of the capacities of those arcs in the cut which are directed from X (the part containing S) to Y (the part containing T).
A **minimum cut** is a cut of smallest possible capacity.

2 Flows and cuts
An important connection between *flows* and *cuts* is:
the value of *any* flow ≤ the capacity of *any* cut.

3 Maximum flows and minimum cuts
The connection between *maximum* flows and *minimum* cuts is:
the value of any *maximum* flow ≤ the capacity of any *minimum* cut.

If we can find a flow with value k, and a cut with capacity k (for the same value of k), then:

- the flow is a maximum flow;
- the cut is a minimum cut.

Theorem 3.1: max-flow min-cut theorem
In any basic network, the value of a maximum flow is equal to the capacity of a minimum cut.

An important consequence of this theorem is that *if you can find a minimum cut in a network, then you can immediately deduce the value of a maximum flow.*

4 Maximum flow algorithm

START with any flow (possibly the zero flow).

Part A: labelling procedure

Start from the source S and proceed along the arcs which allow further flow to unlabelled vertices, chosen alphabetically. Use a forward arc where possible, but if none exists try using an arc carrying a non-zero flow as a backward arc.

Label each vertex with the letter denoting the previous vertex and a number giving the size of the change of flow possible from that vertex.

If a vertex is reached from which no further progress is possible, cross through the current label of that vertex and try backtracking to the vertex previously labelled, and continue from there.

If the sink T becomes labelled, we say we have *breakthrough*; go to Part B.

If there is no breakthrough, STOP:

- the existing flow is a maximum flow;
- a corresponding minimum cut separates S and the currently labelled vertices from the other vertices.

Part B: flow-augmenting procedure

Augment the flow along the path of the labelled vertices $S, \ldots, T$ by the largest flow consistent with the labels.

Remove the labels and return to Part A.

Section 4 Networks with lower and upper capacities

1 In some networks, it is necessary to specify both the maximum and the minimum allowable flow along each arc. The *minimum* allowable flow along an arc is the **lower capacity** of the arc, and the *maximum* allowable flow is the **upper capacity** of the arc. A basic network can be regarded as a network of this type in which all lower capacities are zero.

2 A **flow** in a network with lower and upper capacities is an assignment of non-negative numbers to the arcs in such a way that the following conditions are satisfied:

(a) the **feasibility condition**: the flow along each arc is not less than the lower capacity and is not more than the upper capacity of that arc:

lower capacity ≤ flow ≤ upper capacity;

(b) the **flow conservation law**: the flow into each vertex (other than S or T) is equal to the flow out of it.

(We represent both flows and capacities on the same diagram: the flow appears between the lower capacity and the upper capacity, and the capacities are shown in bold.)

Terms such as the **value of a flow** and a **maximum flow** are defined as before.

3 If we can find a flow in a network with lower and upper capacities, then we can increase this flow to a maximum flow using the maximum flow algorithm to increase the flow along flow-augmenting paths, provided that *when using a backward arc we do not decrease the flow in that arc to a value less than the lower capacity of the arc.*

Finding an initial flow is not always easy, since we can no longer choose the zero flow as the initial flow. There may not even be such a flow.

4 Feasibility

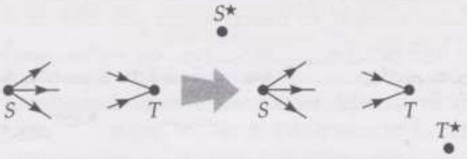
A network N with lower and upper capacities is **feasible** if there exists a flow satisfying all the capacity restrictions. If no such flow exists, then the network is **infeasible**.

The idea behind the feasibility algorithm is to construct from the given network N a basic network $N^\star$ in such a way that if there is a flow in $N^\star$ which satisfies certain conditions (see feasibility theorem below), then the network N is feasible.

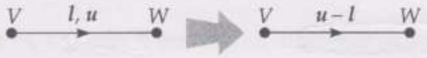
Feasibility algorithm

Construct $N^\star$ from N by the following procedure.

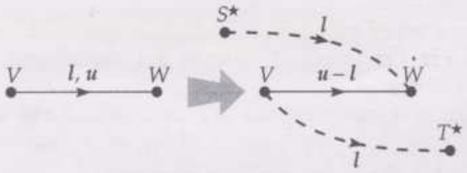
STAGE 1 Take the vertices of $N^\star$ to be the vertices of N, together with two new vertices: a new source $S^\star$ and a new sink $T^\star$.



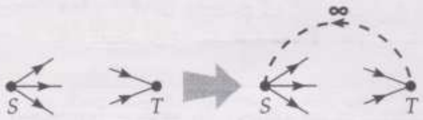
STAGE 2 Replace the lower and upper capacities l and u on each arc VW of N by a single capacity $u - l$ on the corresponding arc of $N^\star$.



STAGE 3 For each arc VW with lower capacity l in N, add in $N^\star$ an arc $S^\star W$ with capacity l and an arc $VT^\star$ with capacity l.



STAGE 4 In $N^\star$, add a new arc from T to S with infinite capacity.



- To show that N is feasible, find a flow in the basic network $N^\star$ in which all the arcs out of $S^\star$ and all the arcs into $T^\star$ are saturated. (Do this by inspection or by using the maximum flow algorithm.)
  Convert this maximum flow in $N^\star$ to a flow in N by 'undoing' the above construction.
  If required, increase the flow in N to a maximum flow. (See item **3**.)
- To show that N is infeasible, show that in $N^\star$ there is no flow from $S^\star$ to $T^\star$ in which all the arcs out of $S^\star$ and all the arcs into $T^\star$ are saturated.

Theorem 4.1: feasibility theorem

A network N with lower and upper capacities is feasible if and only if, in the related basic network $N^\star$, there is a flow from $S^\star$ to $T^\star$ such that all the arcs out of $S^\star$ and all the arcs into $T^\star$ are saturated.

5 Let N be a network with lower and upper capacities, and let C be a cut which separates N into two disjoint parts: X (containing S) and Y (containing T). Then the **capacity of the cut** C is

(the sum of the upper capacities of the arcs of C directed from X to Y)

− (the sum of the lower capacities of the arcs of C directed from Y to X).

(If the lower capacity of each arc is zero, this definition coincides with the definition in Section 3, item **1**.)

6

Theorem 4.2: generalized max-flow min-cut theorem

In any feasible network with lower and upper capacities, the value of a maximum flow is equal to the capacity of a minimum cut.

Design 1 Geometric design

Section 1 Geometric elements

1 The **dimension** of a space is the minimum number of coordinates needed to describe the position of a typical point in that space.

Basic geometric elements in 0, 1, 2 and 3 dimensions are, respectively, *points, line segments, plane segments* and *space segments*. The simplest type of plane segment is a triangle and the simplest type of space segment is a tetrahedron.

2 Convexity

A set S of points in a Euclidean space is **convex** if, for each possible choice of two points P and Q in S, all the points on the line segment joining P and Q also lie in S.

To show that a shape S is *not* convex, it is sufficient to find *one pair* of points P, Q in S such that the line segment PQ does not lie wholly in S.

3 Convex hulls

Let S be a set of points in a Euclidean space. The **convex hull** of S is the smallest convex set containing S; it is denoted by $\langle S \rangle$.

A **polygon** is a shape in the plane bounded by line segments (its *edges*). It consists of all the points on, and enclosed by, the bounding line segments.

Theorem 1.1

Let S be a finite set of points in the plane that do not all lie on a line. Then the convex hull $\langle S \rangle$ is a convex polygon.

A **polyhedron** is an object in space bounded by plane segments (its *faces*). It consists of all the points on, and enclosed by, the bounding plane segments.

Theorem 1.2

Let S be a finite set of points in space that do not all lie in the same plane. Then the convex hull $\langle S \rangle$ is a convex polyhedron.

4 The points and line segments bounding a convex hull can be interpreted as the vertices and edges of a graph, called the **graph** of the convex hull.

5 Simplices

A *single point* is the only type of convex zero-dimensional set, and is called a **0-simplex**.

A *line segment* is the simplest type of convex one-dimensional set, and is called a **1-simplex**.

A *triangle* is the simplest type of convex two-dimensional set, and is called a **2-simplex**.

A *tetrahedron* is the simplest type of convex three-dimensional set, and is called a **3-simplex**.

An **n-simplex**, the simplest type of convex n-dimensional set, is the convex hull of a set of $n + 1$ points, placed in the most general possible positions in n-dimensional space; each line segment between pairs of these points bounds the n-simplex. Its graph is the complete graph K_{n+1}.

6 Orthotopes

The analogues of rectangles and cuboids in higher dimensions are called **orthotopes**.

The **standard n-orthotope** O_n is constructed as follows:

- O_1 is the *unit interval* with endpoints at $x = 0$ and $x = 1$;
- O_2 is the *unit square* with vertices at the points $(0, 0), (1, 0), (0, 1), (1, 1)$;
- O_3 is the *unit cube* with vertices at the points $(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)$.

In general, O_n has 2^n vertices, each with n coordinates, each equal to 0 or 1. The graph of O_n is the **n-cube**.

7 The n-dimensional analogue of a polygon or polyhedron is a **polytope**.

A **cross polytope** is a convex polytope formed by taking the convex hull of the 'cross' formed by going a certain distance in a positive and a negative direction along each coordinate axis. The **standard cross polytope** is constructed by going out a distance 1 along each axis.

In n-dimensional space, a **standard n-dimensional cross polytope** is obtained.

The graph of a cross polytope is formed by drawing an edge between each pair of points except those on the same coordinate axis. In the one-dimensional case there are just two vertices, corresponding to the two points, and no edges; so the graph is the null graph N_2. In the two-dimensional case a square with four vertices each of degree 2 is obtained; so the graph is the cycle graph C_4. In the three-dimensional case an octahedron with six vertices each of degree 4 is obtained, which can be drawn as the standard octahedron graph. The graph of an n-dimensional cross polytope is sometimes called an **n-octahedron**.

8 Spheres

An **n-sphere** is the set of all points up to a given distance away from a given point:

- a 1-sphere is a line segment;
- a 2-sphere is a disc;
- a 3-sphere is a solid sphere (in the usual sense of the word).

Section 2 Planar geometric arrangements

1 A **packing** of shapes or objects is constructed from several copies of a single shape or object arranged to give a regular repeating two-dimensional pattern. If a dot is placed at the centre of each shape or object, a regular array of points, called a **point lattice**, is obtained.

Packings of shapes in the plane include the *triangular, square* and *hexagonal* packing arrangements.

2 A *tile* is a planar shape of finite area with no holes or gaps in it.

A **tiling** (or **tessellation**) of the plane is a covering of the whole plane with tiles in such a way that there are no gaps or overlaps.

Two planar shapes are **congruent** if they are the same size and shape, that is, if one can be superimposed exactly on the other, possibly after turning one over. A given shape **tiles the plane** if the plane can be tiled using only congruent copies of that shape.

A tiling with polygons is an **edge-to-edge tiling** if each edge of each polygon is an edge of another polygon, and no two polygons meet along more than one edge.

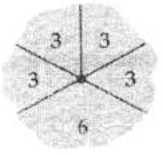
A **regular polygon** is a polygon in which all the edges are of equal length, and the interior angles between each pair of adjacent edges are all equal. A regular polygon is necessarily convex, but a convex polygon need not be regular.

The following table gives the interior angle θ in radians and degrees of an n-sided regular polygon.

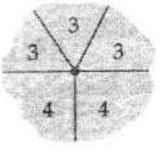
n	3	4	5	6	7	8	9	10	12
θ	$\pi/3$	$\pi/2$	$3\pi/5$	$2\pi/3$	$5\pi/7$	$3\pi/4$	$7\pi/9$	$4\pi/5$	$5\pi/6$
	60°	90°	108°	120°	129°	135°	140°	144°	150°

In any edge-to-edge tiling using regular polygons, the tiles fit together in such a way that each edge belongs to just two tiles and each vertex belongs to at least three tiles.

A **vertex type** is the arrangement of regular polygons around a vertex.



3.3.3.3.6



3.3.3.4.4

3 Regular tilings

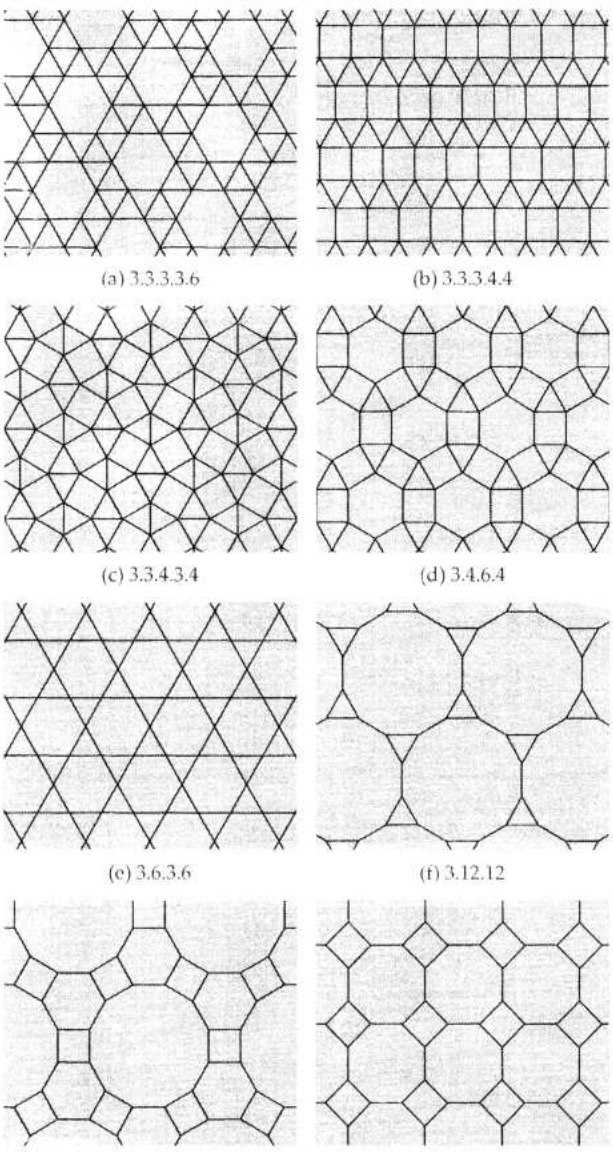
A **regular tiling** is an edge-to-edge tiling with congruent regular polygons.

There are only three types of regular tiling; with triangles, squares and hexagons:

triangular, vertex type: 3.3.3.3.3.3
square, vertex type: 4.4.4.4
hexagonal, vertex type: 6.6.6

4 If two or more types of regular polygon are present at each vertex, there are a further eighteen possible vertex types.

A **semi-regular tiling** is an edge-to-edge tiling with regular polygons, not all congruent, such that each vertex type is the same (up to reflection). There are eight semi-regular tilings:



(a) 3.3.3.3.6 (b) 3.3.3.4.4 (c) 3.3.4.3.4 (d) 3.4.6.4 (e) 3.6.3.6 (f) 3.12.12 (g) 4.6.12 (h) 4.8.8

5 A **demi-regular tiling** is an edge-to-edge tiling with regular polygons in which more than one vertex type occurs. There are infinitely many demi-regular tilings.

6 Dual tilings

Edge-adjacent polygons are polygons that share a common edge.

Given an edge-to-edge tiling with regular polygons, a **dual tiling** is constructed by joining the centre of each polygon to the centres of its edge-adjacent polygons, using line segments.

The **Laves tilings** are the duals of the regular and semi-regular tilings. Three of them are regular tilings — the triangular and hexagonal tilings are duals of each other, and the square tiling is self-dual.

The duals of the eight semi-regular tilings are not regular, semi-regular or demi-regular.

7 A **polygonal animal** is a connected arrangement of non-overlapping regular polygons in edge-to-edge contact. If it comprises exactly *n* polygons, it is called an ***n*-animal**.

formed from	name
equilateral triangles	***n*-iamond**
squares	***n*-omino**
regular hexagons	***n*-hex**

The **graph** of a polygonal animal is formed by placing vertices in the centres of the polygons and joining the vertices in edge-adjacent polygons.

Section 3 Tilings at the Alhambra

1 A **rotation of order *k*** is a rotation through $2\pi/k$ radians or $360/k$ degrees.

Section 4 Polyhedra

1 Regular polyhedra

A **regular polyhedron** is a convex polyhedron in which all the polygonal faces are congruent regular polygons, and in which each vertex has exactly the same arrangement of polygons around it.

There are exactly five regular polyhedra, known as the **Platonic solids**. (See also *Graphs 1*, Section 1, item 5.)

polyhedron	vertices	edges	faces
tetrahedron	4	6	4
octahedron	6	12	8
cube	8	12	6
icosahedron	12	30	20
dodecahedron	20	30	12

The graphs of the regular polyhedra are called the **Platonic graphs**.

2 Dual polyhedra

The **dual** of a convex polyhedron is constructed by placing a new vertex at the centre of each face of the original polyhedron, and joining a pair of new vertices with a line segment whenever the corresponding faces of the original polyhedron are edge-adjacent; thus the roles of the vertices and faces are exchanged.

The duals of all the Platonic solids are Platonic solids.

3 Semi-regular polyhedra

A **semi-regular polyhedron** is a convex polyhedron in which all the polygonal faces are regular polygons, not all congruent, such that each vertex has the same arrangement of polygons around it.

There are infinitely many semi-regular polyhedra comprising two infinite sets (the *prisms* and *antiprisms*) and one finite set of thirteen polyhedra called the **Archimedean solids**.

A **prism** is constructed by taking two parallel congruent regular polygons, aligned one above the other, and joining corresponding vertices with new edges of length equal to that of the sides of the polygons. This results in an 'equatorial strip' of squares with a polygonal top and base.

An **antiprism** is constructed similarly from two parallel congruent regular polygons. The polygons are not aligned: the vertices of the upper polygon are placed above the midpoints of the edges of the lower one. The vertices are joined to produce an 'equatorial strip' of equilateral triangles.

4 Results about polyhedra

Theorem 4.1: Euler's polyhedron formula

Let v, e and f denote, respectively, the numbers of vertices, edges and faces of a convex polyhedron. Then

$$v - e + f = 2.$$

The **degree** of a face of a polyhedron is the number of edges around it.

Theorem 4.2: handshaking lemma for polyhedra

In any polyhedron, the sum of all the face degrees is equal to twice the number of edges.

Theorem 4.3

There are only five regular polyhedra:

- three with triangular faces — the tetrahedron, the octahedron and the icosahedron;
- one with square faces — the cube;
- one with pentagonal faces — the dodecahedron.

Section 5 Incidence structures

1 An **incidence structure** $S(P, L)$ consists of two sets P and L, of different types of object, and an **incidence relation** that shows, for each pair of objects, one from P and one from L, whether or not these two objects are incident.

We often describe the elements of the two sets P and L as 'points' and 'lines', respectively.

The **degree** of a 'point' is the number of 'lines' incident with it, and the **degree** of a 'line' is the number of 'points' incident with it.

An incidence structure is **regular** if every 'point' has the same degree d_p and every 'line' has the same degree d_l.

2 Representing incidence structures

Let $S(P, L)$ be an incidence structure, with n_p 'points' labelled 1 to n_p and n_l 'lines' labelled 1 to n_l. The **incidence matrix** $\mathbf{B}(S)$ of S is the $n_p \times n_l$ matrix in which the entry in row i and column j is 1 if the 'point' labelled i is incident with the 'line' labelled j, and 0 otherwise.

Let $\mathbf{A}$ be any matrix with m rows and n columns. Then the **transpose** of $\mathbf{A}$, denoted by $\mathbf{A}^T$, is the matrix with n rows and m columns obtained by interchanging the rows and columns of $\mathbf{A}$; that is, the entry in row i and column j of $\mathbf{A}$ is the same as the entry in row j and column i of $\mathbf{A}^T$.

3 Dual incidence structures

Let $S(P, L)$ be an incidence structure. Then the incidence structure $S^*(L, P)$ obtained by exchanging the roles of P and L is called the **dual** of $S(P, L)$.

Theorem 5.1

Let S be any incidence structure, and S^* its dual. Then:

(a) the incidence matrix of S^* is the transpose of that of S;

(b) the sum of the degrees of the 'points' of S is equal to the sum of the degrees of its 'lines'.

4 An incidence structure can be represented by a **block table** consisting of a set of columns, each of which represents a 'line' by a list of those 'points' with which it is incident.

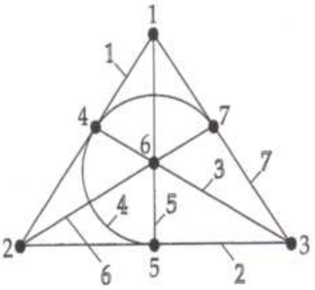
5 Finite projective geometry

A **finite projective plane** is a regular incidence structure for which:

a) for each pair of distinct 'points', there is exactly one 'line' incident with both;

b) for each pair of 'lines' there is exactly one 'point' incident with both.

The **Fano plane** is a finite projective plane with seven points and seven lines, each line being incident with three points, and each point with three lines.



Graphs 2 Trees

Section 1 Tree structures

1 A **tree** is a connected graph that has no cycles.

> **Theorem 1.1: equivalent definitions of a tree**
> Let T be a graph with n vertices. Then the following statements are equivalent.
>
> - T is connected and has no cycles.
> - T has $n - 1$ edges and has no cycles.
> - T is connected and has $n - 1$ edges.
> - T is connected and the removal of any edge disconnects T.
> - Any two vertices of T are connected by exactly one path.
> - T contains no cycles, but the addition of any new edge creates a cycle.

A **rooted tree** (or **branching tree**) is the hierarchical structure in which one vertex is singled out as the starting point, and the branches fan out from this vertex. The starting vertex is the **root**, and is often represented by a small square.

A **sorting tree** is a branching tree that arises when a succession of choices is made, each dependent on the previous one.

Section 2 Counting trees

1 Counting labelled trees

There is a one–one correspondence between labelled trees with n vertices and **Prüfer sequences** — sequences of $n - 2$ numbers $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_{n-2})$, where each $\mathbf{a}_i$ is one of the integers $1, \ldots, n$ (allowing repetition).

To construct a Prüfer sequence from a given labelled tree

STEP 1 Find the vertices of degree 1 and choose the one with the smallest label.

STEP 2 Look at the vertex adjacent to the one just chosen, and place its label in the first available position in the Prüfer sequence.

STEP 3 Remove the vertex chosen in Step 1 and its incident edge, leaving a smaller tree.

Repeat Steps 1–3 for the remaining tree, continuing until there are only two vertices left, then STOP: the required Prüfer sequence has been constructed.

To construct a labelled tree from a given Prüfer sequence containing $n - 2$ numbers

STEP 1 Draw n vertices, labelling them from 1 to n, and make a list of the integers from 1 to n.

STEP 2 Find the smallest number which is in the list but not in the Prüfer sequence, and also find the first number in the sequence; then add an edge joining the vertices with these labels.

STEP 3 Remove the first number found in Step 2 from the list and the second number found in Step 2 from the sequence, leaving a smaller list and a smaller sequence.

Repeat Steps 2 and 3 for the remaining list and sequence, continuing until there are only two terms left in the list. Then join the vertices with these labels and STOP: the required labelled tree has been constructed.

> **Theorem 2.1: Cayley's theorem**
> The number of labelled trees with n vertices is n^{n-2}

2 Counting binary trees

A **binary tree** is a rooted tree in which the number of edges descending from each vertex is at most 2, and a distinction is made between left-hand and right-hand branches.

Let u_n denote the number of binary trees with n vertices. Then $u_1 = 1$, $u_2 = 2$, and for $n \geq 3$,

$$u_n = 2u_{n-1} + (u_1u_{n-2} + u_2u_{n-3} + \cdots + u_{n-2}u_1).$$

3 Central and bicentral trees

To find the 'middle' of a tree, remove all the vertices of degree 1 together with their incident edges, and repeat this process until *either* a single vertex, called the **centre**, *or* two vertices joined by an edge, called the **bicentre**, is obtained. A tree with a centre is called a **central tree**, and a tree with a bicentre is called a **bicentral tree**; every tree is either central or bicentral, but not both.

Section 3 Greedy algorithms

1 Spanning trees

Let G be a connected graph. Then a **spanning tree** in G is a subgraph of G that includes every vertex and is also a tree.

Given a connected graph, a spanning tree may be constructed by using either of the following methods.

Building-up method

Select edges of the graph one at a time, in such a way that no cycles are created; repeat this procedure until all vertices are included.

Cutting-down method

Choose any cycle and remove any one of its edges; repeat this procedure until no cycles are left.

2 Minimum connector problem

Let T be a spanning tree of minimum total weight in a connected weighted graph G. Then T is a **minimum spanning tree** or a **minimum connector** in G.

Kruskal's greedy algorithm for a minimum connector

To construct a minimum spanning tree in a connected weighted graph G, successively choose edges of G of minimum weight in such a way that no cycles are created, until a spanning tree is obtained.

Prim's greedy algorithm for a minimum connector

To construct a minimum spanning tree T in a connected weighted graph G, build up T step by step as follows:

- put an arbitrary vertex from the graph G into T;
- successively add edges of minimum weight joining a vertex already in T to a vertex not in T, until a spanning tree is obtained.

(Note that with Prim's algorithm a *connected* graph is obtained at each stage.)

Tabular (matrix) form of Prim's greedy algorithm for a minimum connector

START with a table of weights for a connected weighted graph, and with no circled entries in the table.

STEP 1 Delete all entries in column 1, and mark row 1 with a star.

STEP 2 Select a smallest entry from the uncircled entries in the row(s) marked with a star.
If no such entry exists, STOP: the edges corresponding to the circled weights form a minimum connector, and its total weight is the sum of the circled weights.
Otherwise, go to Step 3.

STEP 3 (a) circle the weight w_{ij} identified in Step 2;
(b) mark row j with a star;
(c) delete the remaining entries in column j.
Return to Step 2.

Theorem 3.1

Prim's and Kruskal's algorithms always produce a spanning tree of minimum weight.

3 Maximum connector problem

Let T be a spanning tree of maximum total weight in a connected weighted graph G. Then T is a **maximum spanning tree** or a **maximum connector** in G.

Prim's greedy algorithm for a maximum connector

To construct a maximum spanning tree T in a connected weighted graph G, build up T step by step as follows:

- put an arbitrary vertex from the graph G into T;
- successively add edges of maximum weight joining a vertex already in T to a vertex not in T, until a spanning tree is obtained.

Kruskal's greedy algorithm for a maximum connector

Kruskal's algorithm for a minimum connector can be adapted to give a maximum connector.

4 Travelling salesman problem

Given a weighted complete graph, find a minimum-weight Hamiltonian cycle.

There is no known efficient algorithm for the travelling salesman problem — it is an NP-complete problem.

One method for finding an approximate solution to the travelling salesman problem is to find a *lower bound* for the total weight of a minimum-weight Hamiltonian cycle by solving a related minimum connector problem instead.

Method for finding a lower bound for the solution to the travelling salesman problem

STEP 1 Choose a vertex V and remove it from the graph.

STEP 2 Find a minimum spanning tree connecting the remaining vertices, and calculate its total weight W.

STEP 3 Find the two smallest weights, w_1 and w_2, of the edges incident to V.

STEP 4 Calculate the lower bound $W + w_1 + w_2$.

Different choices of V give *different* lower bounds. The *best* lower bound is the *largest*, because it gives the most information about the actual solution.

Section 4 Multi-terminal flows

1 The unit considers the following type of **multi-terminal flow problem**:
find a maximum flow from any location in a given network to any other in the network, assuming that only a single pair of locations can communicate at any one time.

2 Let N be a given undirected network. The **complete network** $C(N)$ is the undirected network constructed by taking the complete graph with the same vertices as N, and assigning to each edge VW a capacity equal to $f(VW)$, the value of a maximum flow between V and W (or W and V).

Theorem 4.1

Let N be an undirected network, and let $C(N)$ be the corresponding complete network. Then, in any cycle of $C(N)$, the smallest capacity occurs more than once.

3 A maximum spanning tree in $C(N)$ can be used to reconstruct the capacities in $C(N)$, using the capacity rule.

Capacity rule

To find $f(VW)$ for two vertices V and W in an undirected network N, look at the path joining the vertices V and W in a maximum spanning tree of the complete network $C(N)$; then $f(VW)$ is the smallest capacity along this path.

Theorem 4.2

Let N be an undirected network with n vertices, and let $C(N)$ be the corresponding complete network. Then $C(N)$ has at most $n - 1$ different capacities.

4 Two undirected networks are **flow-equivalent** if
(a) they have the same number of vertices;
(b) the maximum flows between pairs of vertices in the two networks are the same.

A **flow-equivalent tree** is a **cut tree** if each branch of the tree corresponds to a minimum cut of the original network; that is, each branch, when removed, separates the vertices into two sets X and Y corresponding to a minimum cut in the original network.

5 Algorithm of Gomory and Hu

The algorithm is used for finding the maximum flows between all pairs of vertices in a given undirected network.

The algorithm constructs a cut tree, branch by branch, by calculating a maximum flow for each branch. To attach each new branch to the existing tree, the cut tree property is used: *the minimum cuts in the cut tree and the corresponding minimum cuts in the original network separate the same vertices*. If the original network has n vertices, then a cut tree has $n - 1$ edges, so only $n - 1$ maximum flow calculations are required.

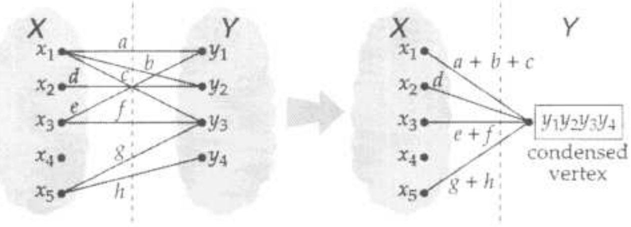
Some of these calculations may be simplified by *condensing* certain vertices of the given network into a single vertex.

Condensing theorem

Consider a minimum cut in an undirected network N separating the vertices into two disjoint sets X and Y.

When finding a maximum flow between any two vertices in X, we may condense all the vertices in Y to a single vertex (and *vice versa*).

Edges connecting a vertex x in X to any vertex in Y are replaced by a single edge with capacity equal to the sum of the capacities of the connecting edges.



Once a cut tree has been constructed, the remaining maximum flows may be found by using the capacity rule, since a cut tree is a maximum spanning tree for the corresponding complete network.

Networks 2 Optimal paths

Section 1 Algorithms using adjacency matrices

1

Theorem 1.1
Let D be a digraph with n vertices labelled 1, 2, ..., n; let $\mathbf{A}$ be its adjacency matrix with respect to this listing of the vertices, and let k be any positive integer. Then the number of walks of length k from vertex i to vertex j is the entry in row i and column j of the matrix $\mathbf{A}^k$.

Theorem 1.2
Let D be a digraph with n vertices labelled 1, 2, ..., n; let $\mathbf{A}$ be its adjacency matrix with respect to this listing of the vertices, and let $\mathbf{B}$ be the matrix

$$\mathbf{B} = \mathbf{A} + \mathbf{A}^2 + \cdots + \mathbf{A}^{n-1}.$$

Then D is strongly connected if and only if each non-diagonal entry in $\mathbf{B}$ is positive; that is, $b_{ij} > 0$ whenever $i \neq j$.

2 Eulerian trails in connected digraphs

An Eulerian trail in a connected digraph is a closed trail that includes every arc just once.

Theorem 1.3
A connected digraph is Eulerian if and only if, for each vertex, the out-degree equals the in-degree.

Theorem 1.4
An Eulerian digraph can be split into cycles, no two of which have an arc in common.

Algorithm for finding an Eulerian trail in a digraph

To find an Eulerian trail in a digraph with no loops, and adjacency matrix **A**, carry out the following steps.

```
START   Set cycle counter $p = 0$.
STEP 1  Increase $p$ by 1.
        Find a row $k$ with a non-zero entry.
        Set $n = k$ and put vertex $v_k$ at the start of a cycle:
        $C_p = v_k \dots$.
STEP 2  Choose $m$ such that $a_{nm} > 0$.
        Include vertex $v_m$ as the next vertex in cycle $C_p$.
        Reduce $a_{nm}$ by 1.
        Set $n = m$.
        If $n \neq k$, return to beginning of Step 2.
        If $n = k$ and not all elements of A have been
        reduced to zero, store cycle $C_p$ and return to
        Step 1; otherwise go to Step 3.
STEP 3  If there is only one stored cycle, this is the
        required Eulerian trail: STOP.
        Otherwise, in the set of stored cycles, find two
        cycles $C_i$ and $C_j$ with at least one vertex $v$ in
        common. In $C_i$, replace the vertex $v$ by $C_j$, after
        writing $C_j$ in the form $C_j = v \dots v$.
        Delete $C_j$ from the set of stored cycles.
        Repeat this process until a single closed trail
        remains — this is the required Eulerian trail:
        STOP.
```

3 Hamiltonian cycles in connected digraphs

A **Hamiltonian cycle** in a connected digraph is a cycle which passes through every vertex.

The algorithm for finding Hamiltonian cycles generates matrices in which each entry is a path or a cycle. The entries are called **strings** and the matrices are combined by *latin multiplication* (denoted by #), which is similar to matrix multiplication with the strings combined as follows:

- string multiplication *concatenates* the strings; for example,
  $v_1v_2v_4 \times v_3v_5 = v_1v_2v_4v_3v_5$
- string additions are written one above the other; for example,
  $v_1v_2v_4 + v_3v_5v_6 = \begin{matrix} v_1v_2v_4 \\ v_3v_5v_6 \end{matrix}$
- a string which includes a vertex more than once is 0 unless such a vertex occurs *just* at the beginning and the end; for example,
  $v_1v_2v_1v_3 = 0$
  $v_1v_2v_3v_1 \neq 0$
- $0 \times (\text{anything}) = 0$
- $0 + (\text{something}) = (\text{something})$

Algorithm for finding Hamiltonian cycles in a digraph

To find all the Hamiltonian cycles in a digraph with n vertices and no loops, carry out the following steps.

```
STEP 1  Define an $n \times n$ matrix C as follows.
        Put
            $c_{ij} = v_iv_j$, if there is an arc from $v_i$ to $v_j$,
            $c_{ij} = 0$, otherwise.
        Define an $n \times n$ matrix D to be the matrix
        obtained from C by deleting the first vertex in
        each non-zero entry of C.
        Set $k = 1$, where $k$ is the power to which the matrix
        C is raised.
STEP 2  Form $\mathbf{C}^{k+1} = \mathbf{C}^k \# \mathbf{D}$, where # denotes latin
        multiplication.
STEP 3  If $k + 1 \neq n$, increase $k$ by 1 and return to Step 2.
        If $k + 1 = n$: STOP.
```

The entries in $\mathbf{C}^k$ give the paths of length k, for $k = 1, \dots, n$. The entries in $\mathbf{C}^n$ give the Hamiltonian cycles.

Section 2 Optimal path algorithms

1 Shortest paths in weighted digraphs

A **shortest path** from a vertex S to a vertex T in a weighted digraph is a path from S to T of smallest possible length.

Shortest path algorithm

This algorithm *looks out from vertices of known potential.*

START Assign potential 0 to S.

GENERAL STEP Consider the vertex (or vertices) just assigned a potential.

For each such vertex V, consider each vertex W which can be reached from V along an arc VW, and assign W the label

(potential of V) + (distance VW)

unless W already has a *smaller* label assigned from an earlier iteration.

When all such vertices W have been labelled, choose the smallest vertex label in the network which is not already a potential, and make it a potential at each vertex where it occurs.

Repeat the general step with the new potential(s).

STOP when T has been assigned a potential.

The shortest distance from S to T is the potential of T.

To find a shortest path, trace backwards from T and include an arc VW wherever

(potential of W) – (potential of V) = distance VW

until S is reached.

2 Longest paths in weighted digraphs

A **longest path** from a vertex S to a vertex T in a weighted digraph is a path from S to T of greatest possible length.

Longest path algorithm

This algorithm *looks back to vertices of known potential.*

START Assign potential 0 to S.

GENERAL STEP Consider all vertices which can be reached directly only from vertices with known potentials.

For each such vertex W, consider each arc VW into W, and assign W the label

(potential of V) + (distance VW)

unless W already has a *larger* label.

When all such arcs VW have been considered, make the vertex label at W a potential.

Repeat the general step with the new potential(s).

STOP when T has been assigned a potential.

The longest distance from S to T is the potential of T. To find a longest path, work backwards from T and include an arc VW wherever

(potential of W) – (potential of V) = distance VW.

If the process does not terminate — for example, if the network has a cycle — there is no longest path

Section 3 Critical path analysis

This section is concerned with scheduling a project involving several activities *when there is no restriction on the number of workers available.*

The **precedence relations** show which activities must precede which.

1 Activity networks

An **activity network** is a network which can be used to schedule a project. It represents the activities from START to FINISH in accordance with the precedence relations.

In an activity network where *vertices represent activities*, each activity is represented by a numbered vertex. An arc joining a vertex X to a vertex Y indicates that the activity represented by X must be completed before the activity represented by Y can be started.

Activity network construction algorithm

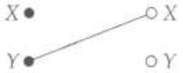
Part A: procedure for numbering the vertices

START Represent each activity by a vertex.

For each vertex, create a **shadow vertex**, so that for each activity there are two corresponding vertices — the original vertex and the shadow vertex.

Construct a bipartite graph in which one set of vertices consists of the original vertices, and the other set consists of the shadow vertices.

If an activity Y must follow an activity X, draw an edge joining the original vertex representing Y to the shadow vertex representing X.



STEP 1 Number consecutively all the original vertices (chosen in any order) which have no edges incident with them. Record the numbering, together with the iteration number.

STEP 2 Delete all numbered vertices, their corresponding shadow vertices, and all edges incident with these vertices.

If not all the vertices have been numbered, return to Step 1.

If all the vertices have been numbered, go to Part B, Step 3.

Part B: procedure for drawing the activity network

STEP 3 Draw a START vertex, and the vertices numbered in the first iteration.

Draw an arc from the START vertex to each vertex which was numbered in the first iteration.

Assign a weight of zero to each arc.

STEP 4 Draw the vertices which were numbered in the next iteration.

To each such vertex Y, draw an arc from each previously numbered vertex X if there is an edge joining the original vertex Y to the shadow vertex X in the original bipartite graph constructed in Part A.

Assign a weight to each arc XY equal to the duration of the activity X.

Repeat until all vertices have been included in the activity network.

STEP 5 Draw a FINISH vertex.

From each terminal vertex Z (that is, each vertex whose out-degree is zero), draw an arc to the FINISH vertex.

Assign a weight to each such arc equal to the duration of the corresponding activity Z. STOP. The activity network has now been constructed.

An **alternative type of activity network** uses arcs to represent activities and vertices to represent *events*. An **event** is a stage in the process corresponding to the start or finish of one or more activities. In this type of activity network, it may be necessary to introduce a *dummy activity* to show that an activity cannot start until another activity has been completed. Dummy activities have no time duration. They are also used when two activities have the same start and end events (each activity can then be referred to by a different number).

2 Critical paths

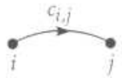
The *minimum completion time* for a project can be calculated from the activity network for the project. To do this, we find a longest path, called a **critical path**, from the START vertex to the FINISH vertex. By a longest path we mean a path for which the sum of the times associated with the arcs of the path (that is, the length of the path) is the greatest possible. The **minimum completion time** is equal to the **length of a critical path**.

The maximum time by which an activity may be delayed without delaying the project is called the **float** of that activity.

Conventions

For a network involving n activities, the START vertex is regarded as the 0th vertex and the FINISH vertex as the $(n+1)$th vertex.

The duration of activity i represented by the arc ij is denoted by $c_{i,j}$.



The algorithm assigns labels p_j and e_j to each vertex j for $j = 0, 1, ..., n+1$. When the algorithm has been completed:

e_j is the length of the longest path from the START vertex to vertex j;

p_j is the number of the preceding vertex on this longest path.

Critical path construction algorithm

Part A: labelling procedure

STEP 1 Assign to the START vertex (vertex 0) the labels $p_0 = 0$ and $e_0 = 0$.

STEP 2 Carry out the following procedure for each vertex j, starting with $j = 1$ and continuing with $j = 2$, $j = 3$, and so on, until all the vertices (including the FINISH vertex, corresponding to $j = n + 1$) have been labelled.

For the current vertex j, calculate, for each arc ij incident to vertex j, the sum

$$e_i + c_{i,j}.$$

Choose the maximum value of these sums for all such arcs ij, and set e_j equal to this value.

Set p_j equal to the value of i for which this sum is largest; in the case of a tie, choose any of the appropriate values.

Part B: tracing back procedure

STEP 3 Start with the FINISH vertex $n+1$, and mark the arc joining this vertex to the preceding vertex p_{n+1}.

STEP 4 Consider the vertex j from which the last marked arc is incident. Mark the arc joining this vertex to the preceding vertex p_j.

Repeat until the START vertex is reached. STOP. The marked arcs form a critical path.

The sum of the weights on the arcs of the critical path is the minimum completion time, and is given by the value of e_{n+1}.

3 Earliest and latest starting times

The critical path construction algorithm assigns a label e_i to each vertex i. The value of e_i is the length of the longest path from the START vertex to the vertex i. For each activity i, e_i is the **earliest starting time** of activity i, since activity i cannot be started until all preceding activities have been completed. For each activity i, the **latest starting time** l_i is the latest time at which activity i can be started without delaying the whole project.

The **float** of activity i (the maximum amount of time by which the activity can be delayed without delaying the project) is the difference between these two times:

float of activity $i = l_i - e_i$.

For an activity i on the critical path, $e_i = l_i$. The float of an activity on the critical path is therefore zero.

Algorithm for calculating latest starting times

This algorithm is applied to an activity network (with n activities) for which the minimum completion time has been calculated. It is used to calculate the value of the latest starting time l_i for each vertex i, for $i = n+1, n, ..., 1, 0$.

STEP 1 For vertex $n+1$ (the FINISH vertex), set l_{n+1} equal to the minimum completion time.

STEP 2 Carry out the following procedure for each vertex i, starting with $i = n$ and continuing with $i = n-1$, $i = n-2$, and so on, until all the vertices including vertex 0 (the START vertex) have been considered.

For the current vertex i, calculate, for each arc ij incident from vertex i, the difference

$$l_j - c_{i,j},$$

where $c_{i,j}$ is the duration of activity i.

Choose the minimum value of these differences for all such arcs ij, and set l_i equal to this value.

When the START vertex has been considered, STOP.

(Note that l_0 must be zero — if this value is not obtained, a mistake has been made somewhere.)

Section 4 Scheduling

1 An ideal schedule satisfies the following conditions, called the **factory rules**.

1. No worker may be idle if there is some activity which can be done.
2. Once a worker starts an activity, that activity must be continued by the same worker until it is completed.
3. The project must be completed as soon as possible with the manpower available.

2 Algorithm for scheduling activities

The algorithm assigns activities to a number of processors, which may be people or machines. It is assumed that the activity network has been constructed and the latest starting times calculated.

The algorithm makes use of a hypothetical clock — called the **project clock** — to keep track of time. At the end of each iteration, the project clock is advanced so that it records the time for which the project has been running.

Critical path scheduling algorithm

START Set the project clock to 0.

STEP 1 If at least one processor is free, assign to any free processor the most critical unassigned activity which can be started. (The most critical activity is one with the smallest latest starting time.)

Repeat until no processor is free or until no unassigned activity can be started.

STEP 2 Advance the project clock until a time is reached when at least one activity has been completed, so that at least one processor is free.

If not all the activities have been assigned, return to Step 1.

If all the activities have been assigned, advance the project clock until all the activities have been completed. STOP: the project clock gives the minimum completion time.

3 Outline of algorithm incorporating a protection scheme for type of example in Section 4.2

STEP 1 Compute for each activity the sum of the earliest starting time and the latest starting time, and rank the activities in ascending order according to the values of these sums.

STEP 2 Assign activities successively to free workers according to this ranking, taking into account the precedence relations. If a preceding activity has not already been assigned, break the ranking order when assigning the activities.

Section 5 Bin packing

A problem to determine the minimum number of workers required to complete a project within a given period is an example of a *bin-packing problem*.

A **bin-packing problem** involves trying to pack items (activities) into the minimum number of 'bins', each of the specified capacity.

Next-fit packing algorithm

Always place the next item to be packed in the current bin if possible; otherwise, place it in the next bin.

First-fit packing algorithm

Always place the next item to be packed in the lowest numbered bin which can accommodate that item.

First-fit decreasing packing algorithm

Order the items in decreasing order of size, and apply the first-fit procedure to this reordered list.

If the items to be packed are available only one at a time, so no reordering of items is possible, an algorithm such as the next-fit algorithm or the first-fit algorithm must be used. Such algorithms are called **on-line algorithms**.

If it is possible to reorder the items to be packed, the first-fit decreasing algorithm can be used. Such an algorithm involving reordering is called an **off-line algorithm**.

Design 2 Kinematic design

Section 1 Kinematic structure

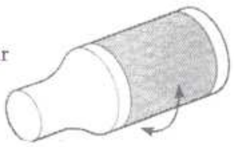
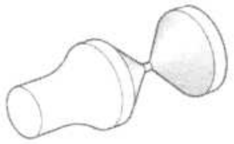
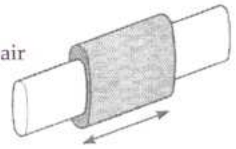
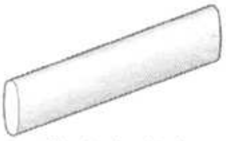
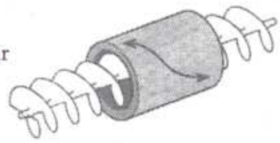
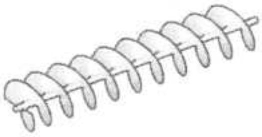
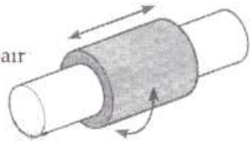
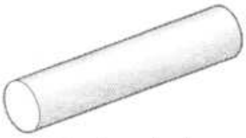
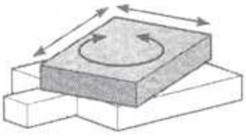
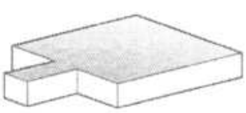
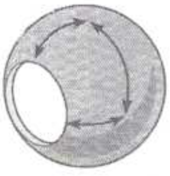
1 A **kinematic system** is a mechanical system designed to produce, transmit, control, constrain, or resist movement.

The design of kinematic systems tries to achieve various *degrees* of movement. In particular, it concerns the *arrangment* and *interconnection* of the component parts of a system so that their motions are suitably controlled and constrained. This is what is meant by **kinematic structure**.

The primitive rigid components of kinematic systems are called **links**. The interconnections between links are called **joints**.

To emphasize the fact that binary joints involve just *two* links, they are referred to as **kinematic pairs**.

The six **Reuleaux pairs** are illustrated below.

pair	sketch	contact surface
revolute pair		 cone
prismatic pair		 elliptical cylinder
screw pair		 helicoid
cylindric pair		 circular cylinder
planar pair		 plane
spherical pair		 sphere

2 The **direct graph** of a kinematic system is a graph in which the vertices represent the joints of the system and the edges represent the links of the system. In a direct graph, two vertices are joined by an edge if the corresponding joints belong to the same link.

The **interchange graph** of a kinematic system is a graph in which the vertices represent the links of the system and the edges represent the joints of the system. In an interchange graph, two vertices are joined by an edge if the corresponding links are connected at a common joint.

3 The combined position and orientation of a link in space, relative to a reference coordinate system, is called its **pose**. If the pose changes with time, then the link is in **motion**.

4 Types of kinematic system

When all the axes of a system are parallel, we refer to the system as **planar** because the links all move in parallel planes. When all the axes intersect in a single point, the system is called a **spherical system**, because the links all move on the surfaces of concentric spheres. If the axes are all mutually skew, the system is called a **spatial system**, because all the links have general motions in space.

5 The **mobility** of a kinematic system is the number of independent quantities or coordinates that must be specified in order to describe the pose of every link of the system with respect to a coordinate frame fixed on a chosen reference link, called the **fixed link**.

Section 2 Braced rectangular frameworks

1 A rectangular framework consists of one or more rectangles, or **bays**.

A **brace** is a mechanical restriction on the motion of a bay in a rectangular framework which reduces the bay's mobility by 1, so that one fewer quantity is required to specify the pose of every link of the framework.

2 In a standard rectangular framework, a **row** is the set of all links forming the vertical sides of a horizontal string of bays, and a **column** is the set of all links forming the horizontal sides of a vertical string of bays.

If the rows are numbered $r_1, r_2, \ldots, r_i, \ldots, r_n$ sequentially from top to bottom and the columns are numbered $c_1, c_2, \ldots, c_j, \ldots, c_m$ sequentially from left to right, then **bay**(i, j) is the bay whose vertical links belong to row r_i and whose horizontal links belong to column c_j.

Theorem 2.1

A braced rectangular framework, with rows $r_1, r_2, \ldots, r_i, \ldots, r_n$ and columns $c_1, c_2, \ldots, c_j, \ldots, c_m$ is rigid if and only if its braces are located such that, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$:

(a) r_i must remain parallel to r_j, for all r_i and r_j;

(b) c_i must remain parallel to c_j, for all c_i and c_j;

(c) r_i must remain perpendicular to c_j, for all r_i and c_j;

under any attempted deformation of the framework.

Theorem 2.2

A braced rectangular framework is rigid if and only if its associated bipartite graph is connected.

A **bracing** of a framework is a particular allocation of braces to bays of the framework.

4 A rigid braced rectangular framework is **minimally braced** if no brace can be removed without destroying the rigidity; the corresponding bracing is a **minimum bracing**.

Theorem 2.3

A rigid braced rectangular framework is minimally braced if and only if its associated bipartite graph is a spanning tree.

Corollary

If the bipartite graph associated with a rigid braced rectangular framework has either of the following properties, then the corresponding bracing is not a minimum bracing:

- the graph has n vertices and more than $n - 1$ edges;
- the graph contains a cycle.

Section 3 Freedom and constraint

1 The **freedoms of a link** are the independent quantities or coordinates that must be specified in order to describe the pose of the link with respect to some reference coordinate frame.

The **freedoms of a two-link kinematic system** are the independent quantities or coordinates that must be specified in order to describe the pose of one link of the system with respect to some reference coordinate frame fixed in the other link of the system.

Theorem 3.1

At least three points must be considered when describing the freedoms of a link in space. Furthermore, once the positions of any three distinct non-collinear points of a link have been specified, then the pose of the whole link has been specified.

2 A **constraint** on a link is a geometric restriction that removes *one* of its freedoms. The pattern is summarized by

$$M = F - C,$$

where M is the mobility, F is the number of freedoms and C is the number of constraints.

3 The **connectivity** of a kinematic pair considered as the only joint in a two-link kinematic system, is the number of freedoms of one of the links with respect to the other.

Section 4 Planar kinematic systems

In this section, attention is restricted to those planar kinematic systems containing only revolute pairs.

Theorem 4.1: first mobility criterion

In a planar kinematic system containing only revolute pairs, the mobility M of the system relative to one of the links (considered as the fixed link) is given by the mobility criterion

$$M = 3(n-1) - 2j,$$

where n is the total number of links (regardless of their multiplicity) and j is the number of revolute pairs.

The value obtained for M from the mobility criterion in Theorem 4.1 is always an integer. There are the following three possible cases.

If M is *positive* and if there are no geometrical complications, then the links of the planar system can have relative motion and the mobility of the joints must be controlled in order to control the motion of the system. The system is said to be **mobile**.

If M is zero and if there are no geometrical complications, then the links of the system have no relative motion and the system is said to be **immobile**.

If M is *negative* and if there are no geometrical complications, then the links of the system again have no relative motion. But, in addition, some of the links and joints can be removed without allowing the remaining links to have any relative motion. The system is said to be **overconstrained**.

Theorem 4.2: second mobility criterion

In a planar kinematic system containing links and joints of various multiplicities, where each r-ary joint is equivalent to $r - 1$ *revolute* pairs, the mobility M of the system relative to one of the links (the fixed link) is given by the mobility criterion

$$M = 2g + 3n_0 + n_1 - ((2r-3)n_r + \cdots + (2s-3)n_s) - 3,$$

where g is the total number of joints (regardless of their multiplicity), no link has multiplicity greater than s, and n_r is the number of r-ary links $(2 \leq r \leq s)$.

Theorem 4.3: mobility criterion for graphs

In a simple connected graph whose vertices all have degree greater than or equal to 2, and which therefore can be considered to be the direct graph of a planar kinematic system containing multiple joints (each of which can be regarded as a combination of revolute pairs) and only binary links, the mobility M of the graph relative to one of the edges (considered to be the *fixed edge*) is given by the mobility criterion

$$M = 2g - n_2 - 3,$$

where g is the total number of vertices of the graph (regardless of their degree) and n_2 is the number of edges of the graph.

As with planar kinematic systems, the value obtained for M from the mobility criterion in Theorem 4.3 is always an integer. Again, M can be positive, zero or negative, and, by analogy with the planar kinematic systems, the *graph* is then said to be **mobile**, **immobile** or **overconstrained**, respectively.

Graphs 3 Planarity and colouring

Section 1 Planarity

1 Planar graphs

A graph G is **planar** if it can be drawn in the plane in such a way that no two edges meet except at a vertex with which they are both incident. Any such drawing is a **plane drawing** of G.

If no plane drawing of G exists, then G is **non-planar**.

$K_{3,3}$ and K_5 are non-planar.

2 Euler's formula

Let G be a planar graph. Then any plane drawing of G divides the set of points of the plane not lying on G into regions, called **faces**; one of these faces is of infinite extent and is called the **infinite face**.

Let G be a planar graph, and let f be any face of a plane drawing of G. Then the **degree of** f, denoted by **deg** f, is the number of edges encountered in a walk around the boundary of the face f.

If all faces have the same degree g, then G is **face-regular of degree** g.

Theorem 1.1: handshaking lemma for planar graphs

In any plane drawing of a planar graph, the sum of all the face degrees is equal to twice the number of edges.

Theorem 1.2: Euler's formula for planar graphs

Let G be a connected planar graph, and let n, m and f denote, respectively, the numbers of vertices, edges and faces in a plane drawing of G. Then

$$n - m + f = 2.$$

Corollary 1.1

Let G be a simple connected planar graph with n (≥ 3) vertices and m edges. Then

$$m \leq 3n - 6.$$

Corollary 1.2

Let G be a simple connected planar graph with n (≥ 3) vertices, m edges and no triangles. Then

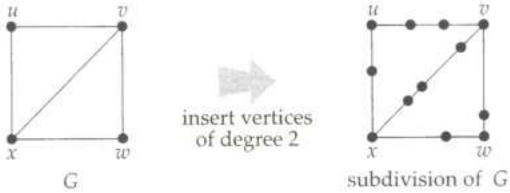
$$m \leq 2n - 4.$$

Corollary 1.3

Let G be a simple connected planar graph. Then G contains a vertex of degree 5 or less.

Corollaries 1.1 and 1.2 can be used to prove (by contradiction) that certain graphs are *non-planar*.

3 Subdivisions and contractions



Any graph formed from a graph G by inserting vertices of degree 2 into the edges of G is called a **subdivision** of G.

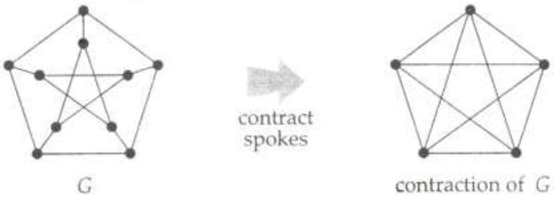
- If G is a planar graph, then every subdivision of G is planar.
- If G is a subdivision of a non-planar graph, then G is non-planar.
- If G is a graph that contains a subdivision of K_5 or $K_{3,3}$, then G is non-planar.
- If G is a non-planar graph, then G contains a subdivision of K_5 or $K_{3,3}$.

Theorem 1.3: Kuratowski's theorem

A graph is planar if and only if it does not contain a subdivision of K_5 or $K_{3,3}$.

Another characterization of planar graphs involves the notion of 'contracting' an edge. This is done by bringing one vertex closer and closer to the other vertex until they coincide, and then coalescing any resulting multiple edges into a single edge.

A **contraction** of a graph is the result of a sequence of edge contractions. For example, K_5 is a contraction of the Petersen graph, since it is the result of contracting each of the five 'spokes'.



Theorem 1.4

A graph is planar if and only if it does not contain a subgraph that has K_5 or $K_{3,3}$ as a contraction.

Theorems 1.3 and 1.4 do not provide an easy way of showing that a given graph is planar, since this would involve looking at a large number of subgraphs and verifying that none of them is a subdivision of K_5 or $K_{3,3}$, or contains K_5 or $K_{3,3}$ as a contraction. For this reason, no currently used algorithm for testing the planarity of a graph is based on either of these theorems.

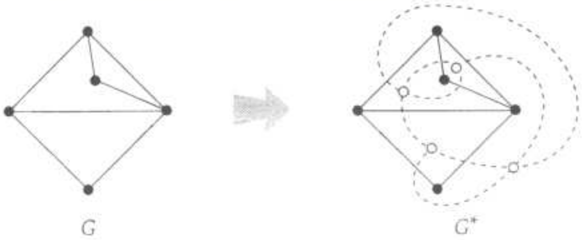
4 Duality

(The following definition assumes that a plane drawing of G is available.)

Let G be a connected planar graph. Then a **dual graph** G^* is constructed from a given plane drawing of G, as follows.

(a) Draw one new vertex in each face of the plane drawing — these are the vertices of G^*.

(b) For each edge e of the plane drawing, draw a line joining the vertices of G^* in the faces on either side of e — these lines are the edges of G^*.



Different plane drawings of a planar graph G may give rise to non-isomorphic dual graphs G^*.

If G is a plane drawing of a connected planar graph, then so is its dual G^*, so $(G^*)^*$, the dual of G^* can be constructed.

The graph $(G^*)^*$ is isomorphic to G.

Theorem 1.5

Let G be a plane drawing of a connected planar graph with n vertices, m edges and f faces. Then G^* has f vertices, m edges and n faces.

Correspondences are given in the following table.

plane drawing G	dual graph G^*
an edge of G	an edge of G^*
a vertex of degree k in G	a face of degree k in G^*
a face of degree k in G	a vertex of degree k in G^*
a cycle of length k in G	a cutset of G^* with k edges
a cutset of G with k edges	a cycle of length k in G^*

These correspondences can be used to obtain new results from old ones by *dualizing*.

Since loops (cycles of length 1) and pairs of multiple edges (cycles of length 2) in G correspond to cutsets with 1 and 2 edges in G^*, Corollary 1.1 can be dualized to give the following theorem.

Theorem 1.6

Let G^* be a connected planar graph with f faces and m edges, and with no cutsets with 1 or 2 edges. Then

$$m \leq 3f - 6.$$

Similarly, Corollary 1.3 can be dualized to give the following theorem.

Theorem 1.7

Let G^* be a connected planar graph with no cutsets with 1 or 2 edges. Then G^* has a face of degree 5 or less.

5 Testing for planarity

- If the graph is disconnected, then it is planar if and only if each of its components is planar.
- If the graph has a cut vertex (a vertex whose removal disconnects the graph), then it is planar if and only if each of the subgraphs obtained when the graph is 'broken apart' at the cut vertex is planar.
- If the graph has loops or multiple edges, then it is planar if and only if the graph obtained by removing the loops and coalescing the multiple edges is planar.

Cycle method for planarity testing

Given a graph G to test for planarity, the idea is to find a Hamiltonian cycle, to draw this cycle as a regular polygon, and then to try to draw the remaining edges so that no edges cross.

Choose a Hamiltonian cycle C. Then list the remaining edges of G and try to divide them into two disjoint sets A and B, as follows:

A is a set of edges which can be drawn *inside* C without crossing;

B is a set of edges which can be drawn *outside* C without crossing.

If this is possible, the graph G is planar, and the sets A and B can be used to obtain a plane drawing of G. If this is not possible, the graph G is non-planar.

Two edges are **incompatible** if they cannot both be drawn inside C, or both be drawn outside C, without crossing. Otherwise they are **compatible**.

Section 2 Colouring maps

Theorem 2.1: four colour theorem for maps

The countries (faces) of any map can be coloured with four (or fewer) colours in such a way that neighbouring countries are coloured differently.

A **map** is a plane drawing of a connected planar graph containing no cutsets with 1 or 2 edges (in particular, a map contains no vertices of degree 1 or 2). A *country* is a face of a map.

Every map must contain at least one of the following:

- a country with just two boundaries — a digon;
- a country with just three boundaries — a triangle;
- a country with just four boundaries — a square;
- a country with five boundaries — a pentagon;
- two adjacent pentagons;
- a pentagon adjacent to a hexagon.

Such a set of configurations is an *unavoidable set*.

A configuration is *reducible* if, when it is removed, the remaining map four-coloured, and then the configuration reinstated, the colouring can always be extended to that configuration.

Theorem 2.2: six colour theorem for maps

The countries of any map can be coloured with six (or fewer) colours in such a way that neighbouring countries are coloured differently.

Theorem 2.3: five colour theorem for maps

The countries of any map can be coloured with five (or fewer) colours in such a way that neighbouring countries are coloured differently.

Section 3 Vertex colourings and decompositions

Attention is restricted to *simple* graphs.

1 Vertex colourings

Let G be a simple graph. A ***k*-colouring** of G is an assignment of k colours to the vertices of G in such a way that adjacent vertices are assigned different colours. If G has a k-colouring, then G is ***k*-colourable**.

The **chromatic number** of G, denoted by $\chi(G)$ is the smallest number k for which G is k-colourable.

Theorem 3.1

Let G be a simple graph whose maximum vertex degree is d. Then

$$\chi(G) \leq d + 1.$$

Theorem 3.2: Brooks' theorem

Let G be a connected simple graph whose maximum vertex degree is d. If G is neither a cycle graph with an odd number of vertices, nor a complete graph, then

$$\chi(G) \leq d.$$

To find the chromatic number $\chi(G)$ of a simple graph G

Try to find an upper bound and a lower bound which are the same; then $\chi(G)$ is equal to this common value. (If $\chi(G) \leq k$ and $\chi(G) \geq k$, then $\chi(G) = k$.)

possible upper bounds for $\chi(G)$

- the number of colours in an explicit vertex colouring of G
- the number n of vertices in G;
- $d + 1$, where d is the maximum vertex degree in G·
- d, where d is the maximum vertex degree in G, provided that G is not C_n (for odd n) or K_n.

possible lower bound for $\chi(G)$

- the number of vertices in the largest complete subgraph in G.

2 Colouring planar graphs

Theorem 3.3: six colour theorem for planar graphs

The vertices of any simple connected planar graph can be coloured with six (or fewer) colours in such a way that adjacent vertices are coloured differently.

Theorem 3.4: five colour theorem for planar graphs

The vertices of any simple connected planar graph can be coloured with five (or fewer) colours in such a way that adjacent vertices are coloured differently.

Theorem 3.5: four colour theorem for planar graphs

The vertices of any simple connected planar graph can be coloured with four (or fewer) colours in such a way that adjacent vertices are coloured differently.

3 Finding a vertex colouring

Greedy algorithm for vertex colouring

```
START  with a graph G and list of colours 1, 2, 3, ...
STEP 1 Label the vertices $a, b, c, \ldots$ in any manner.
STEP 2 Colour the vertex labelled with the earliest letter
       in the alphabet not yet coloured with the first
       colour in the list not used for any adjacent vertex
       already coloured.
```

Repeat Step 2 until all the vertices are coloured, then STOP.

A vertex colouring of G has been obtained. The number of colours used depends on the labelling chosen for the vertices in Step 1.

Theorem 3.6

For any graph G, there is a labelling of the vertices for which the greedy algorithm yields a vertex colouring with $\chi(G)$ colours.

4 Vertex decompositions

A **vertex decomposition** of G is obtained by splitting the set of *vertices* of G into disjoint subsets.

A **dominating set** of vertices in a graph G is a set S of vertices with the property that each vertex of G is either in S or adjacent to a vertex of S.

A **minimum dominating set** is a dominating set of smallest possible size.

The **dominating number** of G, denoted by **dom (G)**, is the number of vertices in a minimum dominating set.

For colouring problems, *in each subset, no two vertices are adjacent*. For domination problems, *each subset contains a vertex adjacent to all the other vertices in that subset.*

The *independence problem for a graph G* is that of finding the largest possible set of vertices of G, no two of which are adjacent.

An **independent set** of vertices in a graph G is a set of vertices of G, no two of which are adjacent.

A **maximum independent set** is an independent set of largest possible size.

The **independence number** of G, denoted by **ind (G)**, is the number of vertices in a maximum independent set.

Theorem 3.7

For any graph G with n vertices:

(a) $\text{dom}(G) \le \text{ind}(G)$;

(b) $\chi(G) \times \text{ind}(G) \ge n$.

Section 4 Edge colourings and decompositions

Attention is restricted to graphs *without loops.*

1 Edge colourings

Let G be a graph without loops. A ***k*-edge colouring** of G is an assignment of k colours to the edges of G in such a way that any two edges meeting at a vertex are assigned different colours. If G has a k-edge colouring, then G is ***k*-edge colourable**.

The **chromatic index** of G, denoted by $\chi'(G)$, is the smallest number k for which G is k-edge colourable.

Theorem 4.1: Vizing's theorem

Let G be a simple graph whose maximum vertex degree is d. Then

$$d \le \chi'(G) \le d+1.$$

Theorem 4.2: Vizing's theorem (extended version)

Let G be a graph whose maximum vertex degree is d, and let h be the maximum number of edges joining a pair of vertices. Then

$$d \le \chi'(G) \le d+h.$$

Theorem 4.3: Shannon's theorem

Let G be a graph whose maximum vertex degree is d. Then

$$d \le \chi'(G) \le 3d/2, \quad \text{if } d \text{ is even;}$$
$$d \le \chi'(G) \le (3d-1)/2, \quad \text{if } d \text{ is odd.}$$

To find the chromatic index $\chi'(G)$ of a graph G without loops

Try to find an upper bound and a lower bound which are the same; then $\chi'(G)$ is equal to this common value.

(If $\chi'(G) \le k$ and $\chi'(G) \ge k$, then $\chi'(G) = k$.)

possible upper bounds for $\chi'(G)$

- the number of colours in an explicit edge colouring of G;
- the number m of edges in G;
- $d+1$, where d is the maximum vertex degree in G, provided that G has no multiple edges;
- $d+h$, where d is the maximum vertex degree in G and h is the maximum number of edges joining a pair of vertices;
- $3d/2$, where d is the maximum vertex degree and d is even;
- $(3d-1)/2$, where d is the maximum vertex degree and d is odd.

possible lower bound for $\chi'(G)$

- d, the maximum vertex degree in G.

2 Classifying graphs

Theorem 4.4

For the complete graph K_n,

$$\chi'(K_n) = \begin{cases} n-1 & \text{if } n \text{ is even;} \\ n & \text{if } n \text{ is odd.} \end{cases}$$

Theorem 4.5: König's theorem

Let G be a bipartite graph whose maximum vertex degree is d. Then

$$\chi'(G) = d.$$

3 Finding an edge colouring

Greedy algorithm for edge colouring

```
START  with a graph G and list of colours 1, 2, 3, ....
STEP 1 Label the edges $a, b, c, \ldots$ in any manner.
STEP 2 Colour the edge labelled with the earliest letter in
       the alphabet not yet coloured with the first colour
       in the list not used for any previously coloured
       edge that meets it at a vertex.
```

Repeat Step 2 until all the edges are coloured, then STOP.

An edge colouring of G has been obtained. The number of colours used depends on the labelling chosen for the edges in Step 1.

Theorem 4.6

For any graph G, there is a labelling of the edges for which the greedy algorithm yields an edge colouring with $\chi'(G)$ colours.

4 Edge decompositions

An **edge decomposition** of G is obtained by splitting the set of *edges* into disjoint subsets.

A **matching** in a graph G is a set of edges of G, no two of which have a vertex in common.

The **thickness** of a graph G, denoted by $t(G)$, is the minimum number of planar graphs that can be superimposed to form G.

Notation

Let a be any positive number. Then $\lfloor a \rfloor$ is the integer obtained by 'rounding a down', and $\lceil a \rceil$ is the integer obtained by 'rounding a up'.

The connection between these functions is given by

$\lceil a/b \rceil = \lfloor (a/b) + (b-1)/b \rfloor$.

Theorem 4.7

Let G be a simple connected graph with n (≥ 3) vertices and m edges. Then

(a) $t(G) \geq \lceil m/(3n-6) \rceil$;

(b) $t(G) \geq \lceil m/(2n-4) \rceil$, if G has no triangles.

Theorem 4.8

(a) $t(K_n) \geq \lfloor (n+7)/6 \rfloor$;

(b) $t(K_{r,s}) \geq \lceil rs/(2r+2s-4) \rceil$.

5 Decomposition into spanning subgraphs

Some problems reduce to that of decomposing a given graph G into the maximum number of connected subgraphs, each of which includes every vertex of the graph. This number is denoted by $s(G)$.

Theorem 4.9

Let G be a connected graph with n vertices. Then $s(G)$ is the largest integer for which the following statement is true:

> for each positive integer $k \leq n$, at least $(k-1) \times s(G)$ edges must be removed in order to disconnect G into k components.

Theorem 4.10

Let G be a connected graph with n vertices and $s(n-1)$ edges. Then G can be decomposed into s spanning trees if and only if

> for each positive integer $k \leq n$, at least $(k-1) \times s$ edges must be removed in order to disconnect G into k components.

Networks 3 Assignment and transportation

1 Matching problem

1 Matchings

The pairing of some or all of the elements of one set with elements of a second set is called a **matching**. In a matching in a graph, the elements are the vertices, and a matching is indicated by thick edges.

A **maximum matching** is a matching with the largest possible number of edges.

2 Marriage problem

Given a set of men, each of whom knows some women from a given set of women, under what conditions is it possible for all the men to marry women they know?

Theorem 1.1: marriage theorem

A necessary and sufficient condition for there to be a solution to the marriage problem (that is, for every man to be able to marry a woman he knows) is:

> for every subset of m men, the m men collectively know at least m women, for all values of m in the range $1 \leq m \leq n$, where n is the total number of men.

The condition given in Theorem 1.1 is known as the **marriage condition**.

3 Modified marriage problem

When it is not possible for all the men to marry, it is natural to ask the following question, called the **modified marriage problem**:

> what is the maximum number of men who can marry women they know?

If the marriage condition is not satisfied, it can be written instead that $n-d$ men can marry women they know if

> (number of women known by each subset of m men) $\geq$ $m-d$, for all values of m in the range $1 \leq m \leq n$, and for some positive integer d.

This is called the **modified marriage condition**.

Theorem 1.2: modified marriage theorem

If a group of n men each know some of a group of women, the maximum number of men who can marry women they know is equal to the minimum value of the expression:

> (number of women known by a subset of m men) + $(n-m)$, for any subset of m men, and for all values of m in the range $1 \leq m \leq n$.

2 Maximum matching problem

1 Alternating paths

Let G be a bipartite graph in which the set of vertices is divided into two disjoint subsets X and Y. An **alternating path** with respect to a matching M in G is a path which satisfies the conditions:

(a) the path joins a vertex x in X to a vertex y in Y;

(b) the initial and final vertices x and y are not incident with an edge in M;

(c) alternate edges of the path are in M, and the other edges are not in M.

If an alternating path with respect to a matching M can be found, then a matching M' which has one more edge than M can be constructed as follows.

To find a new matching with one more edge

Find an alternating path P with respect to an existing matching M.

Form a new matching M' with the following edges:

- the edges of the alternating path P not in the matching M

and

- the edges of the matching M not in the alternating path P.

2 Maximum matching problem

How can a maximum matching in a bipartite graph be found?

The maximum matching algorithm is given opposite.

3 Assignment problem

1 Assignment problem

If a bipartite graph represents a practical problem, some of the matchings may represent more desirable solutions to the problem than others. This fact is expressed by assigning *costs* to the edges of the bipartite graph. These costs are usually displayed in the form of a **cost matrix**. How can a maximum matching with the lowest total cost be found?

The Hungarian algorithm for the assignment problem involves modification of the cost matrix. For a given cost matrix, the graph containing the original vertices, together with the set of edges whose current cost is zero, is called the corresponding **partial graph**.

The Hungarian algorithm for the assignment problem is given on page 28.

Use of dummy vertices

The Hungarian algorithm for the assignment problem applies to a bipartite graph with two sets of vertices X and Y containing the *same* number of elements. Given a problem in which this condition is not satisfied, it is possible to solve the problem by including dummy vertices and corresponding high entries in the cost matrix. The algorithm can then be applied in the usual way.

2 Bottleneck assignment problem

Suppose that a product is to be made on a serial production line which involves a number of activities, each of which must be completed once to produce one item of the product. A number of people are to be assigned to these activities. It is known how long it takes each person to complete each activity. How can the people be assigned to the activities so that the time taken to complete one item of the product is as short as possible?

Algorithm for bottleneck assignment problem

```
START   with a given cost matrix.
        Set $p = 0$.
STEP 1  (a) Set $d$ = smallest non-zero entry in the current
            cost matrix.
        (b) Reduce each non-zero entry in the current
            cost matrix by $d$.
        (c) Increase $p$ by $d$.
STEP 2  Use the matching algorithm to find a maximum
        matching in the bipartite graph.
        If a complete assignment is achieved, STOP: this
        is an optimum assignment and the value of $p$ is
        the required answer.
        Otherwise, return to STEP 1.
```

In simple examples a maximum matching is usually found *by inspection*. In more complicated examples, in Step 2 it is necessary to find an alternating path with respect to the existing matching and use it to obtain a maximum matching.

4 Transportation problem

1 Transportation problem

A manufacturer has a number of factories, each of which can supply a particular product to a number of warehouses. Each factory has a fixed output, and there is a transportation cost involved in sending the product from each factory to each warehouse. To which warehouses should the products of each factory be sent so that the requirements of all the warehouses are satisfied at minimum total cost?

In the bipartite graph for this type of problem, the factories are represented by black vertices, called **supply vertices**. The warehouses are represented by white vertices, called **demand vertices**.

The Hungarian algorithm for the transportation problem is given on page 30.

Use of a dummy vertex

In a transportation problem in which the total supply is greater than the total demand, it is possible to use the Hungarian algorithm by introducing a **dummy vertex**. This represents an artificial warehouse (or other demand vertex) whose demand is chosen to be equal to the excess supply.

In the cost matrix, the transportation costs from each supply vertex to this dummy vertex must be equal. After the algorithm has been used, a flow pattern which is a solution to the original problem can be found by removing all flows to the dummy vertex.

2 Transhipment problem

In the transportation problem it is assumed that the products from each factory are sent directly to one or more warehouses. However, in a practical situation, it may be cheaper to send the products from a factory first to an intermediate point (one of the other factories or warehouses) and then on to the destination warehouse.

The sending of goods via intermediate points is called **transhipment**, and a problem in which transhipment is allowed is called a *transportation problem with transhipment*, or, more simply, a **transhipment problem**. The graphical representation of a transhipment problem is not a bipartite graph, since goods can be sent from factory to factory, or from warehouse to warehouse. However, the graphical representation can be transformed into a bipartite graph by representing the factory or warehouse by two vertices, one representing the factory or warehouse as a supply point, and one representing it as a demand point.

It is necessary to add to all the demands and supplies an amount of demand or supply at least as great as the total demand or supply. The algorithm for the transportation problem is then applied in the usual way. When the procedure of the algorithm has been completed, it is necessary to remove the artificial demands and supplies (and the corresponding artificial flows) to obtain a solution to the original problem.

Algorithm for finding a maximum matching

START with any matching M in a bipartite graph whose sets of vertices are $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_m\}$.

There is no difficulty in finding an initial matching M. If necessary, it can consist of just one edge.

Part A: labelling procedure

STEP 1 Label with (∗) each vertex in X which is not incident with any edge in the current matching M.

If no vertex in X can be labelled, STOP: the current matching is a maximum matching.

Otherwise, go to Step 2.

STEP 2 Choose a newly labelled vertex in X, say x_i, and label with x_i all the unlabelled vertices in Y joined to x_i by an edge NOT IN M.

Repeat this for all newly labelled vertices in X, and then go to Step 3.

STEP 3 Choose a newly labelled vertex in Y, say y_i, and label with (y_i) all unlabelled vertices in X joined to y_i by an edge IN M.

Repeat this for all newly labelled vertices in Y.

Repeat Steps 2 and 3 until

EITHER a vertex in Y which is not incident with an edge in M is labelled (this is called *breakthrough*), in which case go to Part B, Step 4,

OR no such vertex exists, in which case STOP: the current matching is a maximum matching.

Part B: matching improvement procedure

STEP 4 Find an alternating path as follows.

Start at the breakthrough vertex, and go to the vertex indicated by its label. From this vertex, go to the vertex indicated by *its* label, and so on, until a vertex labelled (∗) is reached.

This path P is an alternating path.

STEP 5 Form a new matching from:

- the edges of the current matching M which are NOT IN the alternating path P;
- the edges of the alternating path P which are NOT IN the current matching M.

Remove all labels and return to Part A, Step 1 to see whether the new matching can be improved further.

Summary of algorithm

START with any matching.

Part A: labelling procedure

Label the vertices to identify an alternating path.

If breakthrough is achieved, go to Part B.

If breakthrough is not achieved, STOP: the current matching is a maximum matching.

Part B: matching improvement procedure

Find an alternating path by tracing back through the labels.

Form a new matching from:

- the edges in the current matching NOT IN the alternating path,
- the edges in the alternating path NOT IN the current matching.

Return to Part A.

Hungarian algorithm for the assignment problem

START with no initial matching in a bipartite graph whose sets of vertices are $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$.

Construction of first partial graph

STEP 0a Assign weights to the vertices as follows.

To each vertex of X, assign a weight equal to the lowest cost on any edge incident to that vertex.

Decrease the cost on each edge by the weight on the vertex to which it is incident.

Repeat this procedure for each vertex in Y.

STEP 0b Construct a partial graph consisting of the vertices of the original bipartite graph, together with only those edges whose current cost is zero.

An original edge cost can be thought of as being the total cost of a trip from one vertex to the other. This cost is divided into three parts: the cost of leaving the vertex in X; the cost of the journey (the new edge cost); and the cost of arriving at the vertex in Y.

When Step 0a has been carried out, for any edge, the sum of the weights on the end vertices plus the new cost is equal to the original cost.

Part A: labelling procedure

STEP 1 Label with (*) each vertex in X which is not incident with any edge in the current matching M.

If no such vertex exists, STOP: the current assignment is an optimum assignment.

Otherwise, go to Step 2.

Note that the first time Step 1 is met, the current matching has no edges.

STEP 2 Choose a newly labelled vertex in X, say x_i, and label with (x_i) all the unlabelled vertices in Y joined to x_i by an edge NOT IN M.

Repeat this for all newly labelled vertices in X.

If no vertex in Y is labelled, STOP: the current assignment is an optimum assignment.

Otherwise, go to Step 3.

STEP 3 Choose a newly labelled vertex in Y, say y_i, and label with (y_i) all unlabelled vertices in X joined to y_i by an edge IN M.

Repeat this for all newly labelled vertices in Y.

Repeat Steps 2 and 3 until

EITHER a vertex in Y which is not incident with an edge in M is labelled (this is called *breakthrough*), in which case go to Part B, Step 4,

OR no more labelling is possible, in which case go to Part C, Step 6.

Part B: matching improvement procedure

STEP 4 Find an alternating path as follows.

Start at the breakthrough vertex, and go to the vertex indicated by its label. From this vertex, go to the vertex indicated by *its* label, and so on, until a vertex labelled (*) is reached.

This path P is an alternating path.

STEP 5 Form a new matching from:

- the edges of the current matching M which are NOT IN the alternating path P;
- the edges of the alternating path P which are NOT IN the current matching M.

Remove all labels and return to Part A, Step 1 to see whether the new matching can be improved further.

Part C: modification of partial graph procedure

STEP 6 Find the lowest current cost on any edge of the original bipartite graph which

- starts at a LABELLED vertex in X;
- ends at an UNLABELLED vertex in Y;

- has a non-zero current cost.

Call this lowest cost δ.

STEP 7 (a) Increase the weight on each labelled vertex in X by δ.

(b) Decrease the weight on each labelled vertex in Y by δ.

(c) For each edge of the original bipartite graph which joins a labelled vertex in X to an unlabelled vertex in Y, decrease the current cost by δ.

(d) For each edge of the original bipartite graph which joins an unlabelled vertex in X to a labelled vertex in Y, increase the current cost by δ.

Part (c) above will have produced at least one more edge with current cost zero. Incorporate all such edges with zero cost in the partial graph, delete all labels, and return to Part A, Step 1.

Note that in Step 7 no negative costs are produced in the cost matrix, since δ was chosen as the lowest non-zero cost. Also, the total cost of a 'trip' from a vertex in X to a vertex in Y via any edge not in the partial graph, that is, the sum of the two vertex weights and the edge cost, is kept constant throughout Step 7.

Summary of algorithm

START with no matching.

Assign weights to the vertices and construct the first partial graph.

Part A: labelling procedure

Label the vertices to identify an alternating path.

If none of the vertices on one side of the graph can be labelled, STOP: the current assignment is an optimum assignment.

If breakthrough is achieved, go to Part B.

If breakthrough is not achieved, go to Part C.

Part B: matching improvement procedure

Find an alternating path by tracing back through the labels.

Form a new matching.

Return to Part A.

Part C: modification of partial graph procedure

Construct a revised cost matrix as follows.

On the existing cost matrix:

draw a *horizontal* line through each labelled vertex in X;

draw a *vertical* line through each labelled vertex in Y;

- find the smallest non-zero entry δ with ONLY a *horizontal* line through it;
- *decrease* all entries with ONLY a *horizontal* line through them by δ; *increase* the weight on the corresponding vertices in X by δ;
- *increase* all entries with ONLY a *vertical* line through them by δ; *decrease* the weight on the corresponding vertices in Y by δ.

Construct a revised partial graph.

Return to Part A.

Hungarian algorithm for the transportation problem

START with no flow in a bipartite graph whose vertices are divided into two sets: supply vertices $A_1, \ldots, A_n$ and demand vertices $b_1, \ldots, b_m$.

Construction of the initial partial graph

STEP 0a Assign weights to the supply and demand vertices as follows.

To each supply vertex, assign a weight equal to the lowest cost on any edge incident with that vertex.

Decrease the cost on each edge by the weight on the supply vertex with which it is incident.

Repeat this for each demand vertex.

STEP 0b Construct a partial graph consisting of the vertices of the original bipartite graph, together with only those edges whose current cost is zero.

Part A: labelling procedure

STEP 1 Label with (*) each supply vertex whose supply has not all been allocated to edges.

If no such vertex exists, STOP: the current solution is a minimum-cost solution.

Otherwise, go to Step 2.

STEP 2 Choose a newly labelled supply vertex A_i, and label with (A_i) all unlabelled demand vertices which are joined by an edge of the partial graph.

Repeat this for all newly labelled supply vertices, and then go to Step 3.

STEP 3 Choose a newly labelled demand vertex b_j, and label with (b_j) all unlabelled supply vertices which are joined to b_j by an edge which has been allocated a flow. STOP:

Repeat this for all newly labelled demand vertices.

Repeat Steps 2 and 3 until

EITHER a demand vertex whose demand is not satisfied is labelled (this is called *breakthrough*), in which case go to Part B, Step 4,

OR no more labelling is possible, in which case go to Part C, Step 6.

Part B: flow-augmenting procedure

STEP 4 Start with the breakthrough vertex, and trace back through the labels until a supply vertex which is labelled (*) is reached. A flow-augmenting path has been found.

The flow-augmenting path plays a similar role in this algorithm to that of the alternating path in the algorithm for the assignment problem.

Calculate the maximum flow F which can be sent along this path as follows.

First, find the minimum of all flows assigned to EVEN edges of the path (that is, the 2nd, 4th, … edges). This is the maximum possible backward flow.

Next, find the minimum value of:

- the available supply at the start vertex of the path;
- the required demand at the end vertex;
- the maximum backward flow.

This is the required value of F.

Note that for a flow-augmenting path with just one edge, there is no backward flow, so we just take the minimum of the supply and the demand.

STEP 5 Form a new flow pattern as follows.

(i) Increase the flows on the ODD edges of the path by this value of flow, F.

Decrease the flows on the EVEN edges by F.

(ii) Decrease the available supply at the START vertex by F.
Decrease the demand at the END vertex by F.

Remove all the labels and return to Part A, Step 1.

Part C: modification of partial graph procedure

STEP 6 Find the lowest current cost on any edge of the original bipartite graph which

- starts at a LABELLED supply vertex;
- ends at an UNLABELLED demand vertex;
- has a non-zero current cost.

Call this lowest cost δ.

STEP 7 (a) Increase the weight on each labelled supply vertex by δ.

(b) Decrease the weight on each labelled demand vertex by δ.

(c) For each edge which joins a labelled supply vertex to an unlabelled demand vertex, decrease the current cost by δ.

(d) For each edge which joins an unlabelled supply vertex to a labelled demand vertex, increase the current cost by δ.

Incorporate all edges which now have a current cost of zero in the partial graph, delete all labels, and return to Part A, Step 1.

Summary of algorithm

START with no flow.

Construct the initial partial graph.

Part A: labelling procedure

Label the vertices to identify a flow-augmenting path.

If no labelling is possible, STOP: the current solution is a minimum-cost solution.

If breakthrough is achieved, go to Part B.

If breakthrough is not achieved, go to Part C.

Part B: flow-augmenting procedure

Find a flow-augmenting path by tracing back through the labels.

Augment the flow.

Return to Part A.

Part C: modification of the partial graph procedure

Construct a new revised cost matrix as follows.

On the existing cost matrix:

draw a *horizontal* line through each labelled *supply* vertex;

draw a *vertical* line through each labelled *demand* vertex:

- find the smallest non-zero entry δ with only a *horizontal* line through it;
- *decrease* all entries with only a *horizontal* line through them by δ;
increase the weight on the corresponding *supply* vertices by δ;
increase all entries with only a *vertical* line through them by δ;
decrease the weight on the corresponding *demand* vertices by δ.

Construct a revised partial graph.

Return to Part A.

Design 3 Design of codes

Section 1 Error detection and correction

1 A **code** is a set of binary words, each containing the same number of **bits** (binary digits 0 and 1). Each binary word is a **codeword**. Such a code is known as a **block code**, because each codeword is a block of binary digits. The number n of bits in each codeword of a block code is the **length** of the code.

2 Most of the codes studied in this unit are constructed by adding $n-k$ **check bits** to each message of length k to make a binary word of length n. Such a code consists of 2^k of the 2^n possible strings of n bits.

A code of length n, with k message bits, is an **(n, k) code**. The number k is the **dimension** of the code.

The **rate** of a code is the number of message bits in each codeword divided by the length of the code. The rate of an (n, k) code is k/n.

3 **Encoding** means identifying each possible binary message of (say) k bits with a binary codeword containing n bits, where $n > k$.

Decoding means estimating the original message from the received word. If few errors have occurred in transmission and the code is efficient, this can be done accurately.

4 An encoding rule that results in the k bits of the message appearing in fixed positions in the corresponding codeword is a **systematic encoding rule**. The k message bits need not necessarily form the first k bits of each codeword, nor need they appear in k consecutive places, but they must appear in the same order in each codeword, in some fixed set of k places.

5 The **weight** $w(\mathbf{x})$ of a binary word $\mathbf{x}$ is the number of 1s in $\mathbf{x}$.

The **even-weight code** of length 4 is the code

{0000, 0011, 0101, 0110, 1001, 1010, 1100, 1111}.

The fourth bit is an **overall parity check**. The term **even-weight** means that the sum of the bits in each codeword is even.

6 **Majority-logic decoding** is used when there are several 'votes' for each bit of information; the value with the largest number of votes is chosen.

7 **Repetition codes**

The **n-fold repetition code $R(n)$** encodes the messages 0 and 1 as follows:

message	*codeword*
0	00 ... 0 (n bits)
1	11 ... 1 (n bits)

The first of these words is the **zero word**, denoted by $\mathbf{0}$. The rate of the code $R(n)$ is $1/n$.

8 The **Hamming distance** $d(\mathbf{x}, \mathbf{y})$ between two binary words $\mathbf{x}$ and $\mathbf{y}$ of the same length is the number of places in which their bits differ.

The Hamming distance possesses the following properties.

For any binary words $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ of the same length:

- $d(\mathbf{x}, \mathbf{y}) \geq 0$;
- $d(\mathbf{x}, \mathbf{y}) = 0$ if and only if $\mathbf{x} = \mathbf{y}$;
- $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$; (symmetry)
- $d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$. (triangle inequality)

9 The **minimum distance** δ of a code is the smallest Hamming distance between two distinct codewords.

An (n, k) code with minimum distance δ is sometimes referred to as an **(n, k, δ) code**.

Theorem 1.1

Let C be a code with minimum distance δ.

If δ is odd, then C can detect and correct up to $(\delta-1)/2$ errors.

If δ is even, then C can detect up to $\delta/2$ errors and correct up to $(\delta-2)/2$ errors.

A code with minimum distance δ can correct $\lfloor(\delta-1)/2\rfloor$ errors, where $\lfloor x \rfloor$ denotes the integer part of x.

Section 2 Linear codes

1 A **linear code** is a code in which the sum (modulo 2) of any two codewords is also a codeword — that is, whenever $\mathbf{x}$ and $\mathbf{y}$ are codewords, then so is $\mathbf{x}+\mathbf{y}$.

2 The **weight** $w(\mathbf{x})$ of a binary word $\mathbf{x}$ is the number of 1s in $\mathbf{x}$.

Theorem 2.1

For a linear code, the weight w possesses the properties:

(a) for each pair of codewords $\mathbf{x}$ and $\mathbf{y}$,
$d(\mathbf{x}, \mathbf{y}) = w(\mathbf{x}+\mathbf{y})$;

(b) the minimum distance δ of the code is the smallest non-zero weight among the codewords.

Other useful properties are:

- $w(\mathbf{x}+\mathbf{y}) \leq w(\mathbf{x}) + w(\mathbf{y})$;
- $w(\mathbf{x}+\mathbf{y}) \geq w(\mathbf{x}) - w(\mathbf{y})$.

3 **Generator sets and matrices**

A **generator set** for a code is a minimal set of codewords with the property that every codeword can be obtained by adding codewords in this minimal set. A generator set is denoted by angled brackets.

For a code with dimension k, any generator set consists of k codewords, none of which can be formed by adding any of the others in the set.

A code can have several different generator sets.

A $k \times n$ matrix $\mathbf{G}$ is a **generator matrix** for an (n, k) linear code C if the binary words that can be expressed as a sum of a subset of the rows of $\mathbf{G}$ are exactly the codewords of C.

A code can have several different generator matrices.

A generator matrix $\mathbf{G}$ can be used to specify a simple rule for assigning a unique codeword to each possible message of k bits.

Let a message $\mathbf{m}$ be a binary word of length k written as a row vector. The matrix product $\mathbf{mG}$ is a binary word of length n, which must be a codeword since $\mathbf{mG}$ is a sum of rows of $\mathbf{G}$. If the bits $m_{j_1}, m_{j_2}, \ldots, m_{j_t}$ of $\mathbf{m}$ are 1 and all the other bits of $\mathbf{m}$ are 0, then $\mathbf{mG}$ can be obtained by adding rows $\mathbf{j}_1, \mathbf{j}_2, \ldots, \mathbf{j}_t$ of $\mathbf{G}$. It can be shown that 2^k different binary words of length n can be obtained by adding rows of $\mathbf{G}$, and hence that no two distinct messages $\mathbf{m}$ of length k can give rise to the same codeword $\mathbf{mG}$.

When calculating the matrix product $\mathbf{mG}$, addition and multiplication modulo 2 are used. The rules are summarized as follows.

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

Encoding using a $k \times n$ generator matrix $\mathbf{G}$ is systematic whenever k of the n columns of $\mathbf{G}$ are distinct columns of the $k \times k$ identity matrix $\mathbf{I}_k$. If column $\mathbf{i}$ of $\mathbf{I}_k$ forms column $\mathbf{j}_i$ of $\mathbf{G}$, then the ith bit of each message appears as the j_i th bit of the corresponding codeword. A $k \times n$ generator matrix is in **standard form** when the first k columns form the $k \times k$ identity matrix $\mathbf{I}_k$.

4 Parity check matrices

Let C be an (n, k) linear code.

A **parity check matrix H** is an $(n-k) \times n$ matrix with the property that the codewords of C are precisely the binary words $\mathbf{x}$ that satisfy the matrix equation $\mathbf{Hx}^T = \mathbf{0}^T$.

A code can have several different parity check matrices.

A binary word $\mathbf{x}$ satisfies $\mathbf{Hx}^T = \mathbf{0}^T$ if and only if the sums (obtained by ordinary addition) of certain of its bits (determined by $\mathbf{H}$) are even — that is, they have even parity. The $n-k$ linear equations are **parity check equations** of the code.

Suppose that a codeword $\mathbf{x}$ from a code with parity check matrix $\mathbf{H}$ is transmitted over a communication channel. Let $\mathbf{e}$ denote the error word that affects $\mathbf{x}$, so that the word $\mathbf{r} = \mathbf{x} + \mathbf{e}$ is received. Then

$$\mathbf{Hr}^T = \mathbf{H}(\mathbf{x}+\mathbf{e})^T = \mathbf{H}(\mathbf{x}^T + \mathbf{e}^T) = \mathbf{Hx}^T + \mathbf{He}^T = \mathbf{0}^T + \mathbf{He}^T$$
$$= \mathbf{He}^T.$$

In other words, the column vector of length $n-k$ that results from premultiplying the transpose of the received word $\mathbf{r}$ by a parity check matrix $\mathbf{H}$ is independent of the codeword transmitted. It depends only on the errors that have occurred. It is the sum of those columns of $\mathbf{H}$ that correspond to the corrupted bits.

For an (n, k) code with parity check matrix $\mathbf{H}$, the **error syndrome** of a received word $\mathbf{r}$ is the column vector $\mathbf{Hr}^T$ of length $n-k$.

5 Hamming codes

Consider an (n, k) code that corrects a single error, and let $m = n-k$. Then a parity check matrix of this code has $2^m - 1$ possible columns. Let $\mathbf{H}$ be the matrix formed from these columns, written in increasing binary order. Then $\mathbf{H}$ is a parity check matrix of a **Hamming code**.

This code has a simple decoding algorithm. If a transmitted codeword is received with its ith bit in error, then the syndrome calculated from the received word is the ith column of $\mathbf{H}$. However, reading from top to bottom, the ith column of $\mathbf{H}$ is the binary representation of the number i. The bit that is in error can therefore be corrected immediately.

It is possible to construct a Hamming code of length $2^m - 1$ for each integer m, where $m \geq 2$. A parity check matrix has m rows, and $2^m - 1$ columns. The code has $2^m - 1 - m$ message bits. All Hamming codes have minimum distance 3, so each is a $(2^m - 1, 2^m - 1 - m, 3)$ code, for some integer m.

6 Perfect codes

Suppose that $\mathbf{c}$ is a codeword in an (n, k, δ) code, and that t is an integer such that $t \leq (\delta-1)/2$ if δ is odd, and $t \leq (\delta-2)/2$ if δ is even. Then there are at least

$$2^k\left(1+\binom{n}{1}+\binom{n}{2}+\cdots+\binom{n}{t}\right)$$

n-bit binary words. However, the total number of n-bit binary words is 2^n, and so

$$1+\binom{n}{1}+\binom{n}{2}+\cdots+\binom{n}{t} \leq 2^{n-k}.$$

This inequality is called the **Hamming inequality;** the upper bound given by this inequality is the **Hamming bound**.

A **perfect code** is a linear code for which the Hamming bound is attained.

Every Hamming code is a perfect code.

Section 3 New codes from old

1 Equivalent codes

Two codes are **equivalent** if the codewords of one can be obtained by rearranging the bits of each codeword of the other according to some fixed rule.

> **Theorem 3.1**
>
> Any $(2^m - 1, 2^m - 1 - m, 3)$ linear code is equivalent to the Hamming code of the same length and dimension.

For this reason, *any* $(2^m - 1, 2^m - 1 - m, 3)$ linear code is known as a *Hamming code.*

> **Theorem 3.2:** Every (n, k) code is equivalent to [~~a code with a systematic encoding rule – that is~~] a code whose parity check matrix can be written in the **standard form** $\mathbf{H}=[\mathbf{A}\,|\,\mathbf{I}]$, where $\mathbf{A}$ is an $(n-k)\times k$ matrix of 0s and 1s and $\mathbf{I}$ is the $(n-k) \times (n-k)$ identity matrix.

If a parity check matrix $\mathbf{H}$ of a code C is expressed in standard form $\mathbf{H} = [\mathbf{A}\,|\,\mathbf{I}]$, then the matrix $\mathbf{G} = [\mathbf{I}\,|\,\mathbf{A}^T]$ is a generator for C. [And conversely, if the generator matrix $\mathbf{G} = [\mathbf{I}\,|\,\mathbf{A}^T]$, then the matrix $\mathbf{H} = [\mathbf{A}\,|\,\mathbf{I}]$ is a parity check matrix for C.]

A **cyclic code** is a code of length n with the property that, whenever $\mathbf{x} = x_1\ x_2\ \ldots x_n$ is a codeword, then so also is $\mathbf{x}' = x_2 \ldots x_n\, x_1$. For each positive integer m, there is a cyclic Hamming code of length $2^m - 1$. In other words, each Hamming code is equivalent to a cyclic Hamming code.

3 Dual codes

Let $\mathbf{H}$ be a parity check matrix of a linear code C. The sum of each possible subset of rows of $\mathbf{H}$ is a unique binary word. Thus exactly 2^{n-k} different words of length n can be obtained from the rows of $\mathbf{H}$. These words are therefore the codewords of an $(n, n-k)$ code, called the **dual code** C^* of the original code C.

A **simplex code** is the dual of a Hamming code: the dual of a $(2^m - 1,\ 2^m - 1 - m,\ 3)$ Hamming code is a $(2^m - 1, m, 2^{m-1})$ simplex code.

The simplex code of length $2^m - 1$ contains the zero word and $2^m - 1$ words of weight 2^{m-1}. If the codewords are considered as vertices of the n-cube, they form a *regular n-simplex* — that is, a simplex with edges of the same length.

Let $\mathbf{H}$ be a parity check matrix of an (n, k) linear code C. The **dual** of C is the $(n, n-k)$ linear code C^* with generator matrix $\mathbf{H}$.

> **Theorem 3.3: duality theorem**
>
> Let C be a linear code. Then $(C^*)^* = C$.

> **Corollary**
>
> Let C be a linear code. Then a parity check matrix of the dual code C^* is a generator matrix for C.

> **Theorem 3.4**
>
> The dual of a cyclic code is cyclic.

4 Extended codes

A given (n, k, δ) code can be extended by adding an overall parity check. An extra bsit is added to each codeword in such a way that the resulting binary word of length $n + 1$ has even weight, so a 0 is added to each codeword of even weight, and a 1 to each codeword of odd weight. The minimum distance of the **extended code** is $\delta + 1$ if δ is odd, and δif δ is even.

A parity check matrix for the code obtained by extending an (n, k) code with parity check matrix **H** is

$$\left[\begin{array}{c|cccc} 1 & 1 & 1 & \dots & 1 \\ \hline 0 & & & & \\ 0 & & \mathbf{H} & & \\ 0 & & & & \end{array}\right].$$

(Here we have added the overall parity check as the *first* bit of each codeword.)

5 The [a | a + b] construction

Theorem 3.5

Let A be an (n, k_A, δ_A) code and B be an (n, k_B, δ_B) code, and let δ be the smaller of $2\delta_A$ and δ_B . Then the code C whose codewords are all the binary words of the form $[\mathbf{a} \mid \mathbf{a} + \mathbf{b}]$, where $\mathbf{a}$ is a codeword of code A and $\mathbf{b}$ is a codeword of code B, is a $(2n, k_A + k_B, \delta)$ code.

If $\mathbf{G}_A$ is a generator matrix for code A and $\mathbf{G}_B$ is a generator matrix for code B, then a generator matrix for code C is

$$\mathbf{G} = \left[\begin{array}{c|c} \mathbf{G}_A & \mathbf{G}_A \\ \hline \mathbf{0} & \mathbf{G}_B \end{array}\right]$$

If $\mathbf{H}_A$ is a parity check matrix for code A and $\mathbf{H}_B$ is a parity check matrix for code B, then a parity check matrix for code C is

$$\mathbf{H} = \left[\begin{array}{c|c} \mathbf{H}_A & \mathbf{0} \\ \hline \mathbf{H}_B & \mathbf{H}_B \end{array}\right]$$

Section 4 First-order Reed-Muller codes

1 Three special codes

$\mathcal{R}(1)$ is the code {00, 01, 10, 11}.

$\mathcal{R}(2)$ is the even-weight code of length 4.

$\mathcal{R}(3)$ is the (8, 4, 4) extended Hamming code.

2 Reed-Muller codes

A **Boolean variable** is a variable that can take only two values, usually 0 and 1. A **Boolean function** is a function of Boolean variables, which can itself take only the same two values.

The set of binary words that correspond to all the Boolean functions of degree at most 1 in m variables, $v_1, v_2, \dots, v_m$, is called the **first-order Reed-Muller code of length 2^m**; it is denoted by $\mathcal{R}(m)$.

The dimension of $\mathcal{R}(m)$ is $m+1$.

The code $\mathcal{R}(m)$ has:

one codeword of weight 0,

$2^{m+1} - 2$ codewords of weight 2^{m-1},

one codeword of weight 2^m .

The minimum distance of $\mathcal{R}(m)$ is 2^{m-1} .

The code $\mathcal{R}(m)$ is a $(2^m, m+1, 2^{m-1})$ code.

Theorem 4.1

The code $\mathcal{R}(m+1)$ consists of all binary words of the form $[\mathbf{a} \mid \mathbf{a} + \mathbf{b}]$, where $\mathbf{a}$ is a codeword in $\mathcal{R}(m)$ and $\mathbf{b}$ is a codeword in the repetition code $R(2^m)$.

A generator matrix for $\mathcal{R}(3)$ is

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Let $\mathbf{G}_1$ be a generator matrix for $\mathcal{R}(m)$. Then a generator matrix for $\mathcal{R}(m+1)$ is

$$\mathbf{G} = \left[\begin{array}{c|c} \mathbf{G}_1 & \mathbf{G}_1 \\ \hline 0\dots0 & 1\dots1 \end{array}\right]$$

First-order Reed-Muller codes can be decoded using a form of majority logic.

Suppose a codeword is transmitted that corresponds to the message $\mathbf{a}$, encoded using the generator matrix $\mathbf{G}$. Let $\mathbf{r}$ be the word received. If no error has occurred in transmission, then $\mathbf{r} = \mathbf{aG}$. This implies relationships between the bits of $\mathbf{r}$ and $\mathbf{a}$ which give several 'votes' for each bit of $\mathbf{a}$.

3 Golay codes

Let A denote the first-order Reed-Muller code $\mathcal{R}(3)$ of length 8 with generator matrix

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

If the order of the first seven bits of each codeword of code A is reversed, the codewords of an equivalent first-order Reed-Muller code of length 8 are obtained. Denote this second code by B.

The **extended Golay code $G(24)$** is the code whose codewords are all the binary words that can be written in the form

$$[\mathbf{a}_1 + \mathbf{b} \mid \mathbf{a}_2 + \mathbf{b} \mid \mathbf{a}_1 + \mathbf{a}_2 + \mathbf{b}],$$

where $\mathbf{a}_1$ and $\mathbf{a}_2$ are codewords in A, and $\mathbf{b}$ is a codeword in B.

The code $G(24)$ is a (24, 12, 8) code.

Let $\mathbf{G}_A$ and $\mathbf{G}_B$ be generator matrices of codes A and B, respectively. Then a generator matrix for the code $G(24)$ is the 12×24 matrix

$$\mathbf{G} = \left[\begin{array}{c|c|c} \mathbf{G}_A & \mathbf{0} & \mathbf{G}_A \\ \hline \mathbf{0} & \mathbf{G}_A & \mathbf{G}_A \\ \hline \mathbf{G}_B & \mathbf{G}_B & \mathbf{G}_B \end{array}\right].$$

The code $G(24)$ is self-dual.

The **Golay code $G(23)$** is a (23, 12, 7) code obtained from $G(24)$ by deleting the last bit of each codeword.

The perfect codes are: the repetition codes $R(n)$ for all odd values of n, the Hamming codes, and the two Golay codes. There are no others.

4 Mariner 9 code

The Mariner 9 code is the code $\mathcal{R}(5)$.

Graphs 4 Graphs and computing

Section 1 Efficiency of algorithms

1 The **time complexity function** $T(n)$ for an algorithm gives the maximum time taken to process any input of size n, at 1 time unit per comparison made.

2 The **space complexity function** $S(n)$ for an algorithm gives the maximum space required in a computer's memory for storing the data used by the algorithm.

3 Let $T(n)$ be a time complexity function, and let $g(n)$ be a function for which there exists both a positive constant c and a (large enough) number N such that

$$T(n) \leq c.g(n), \quad \text{for all } n \geq N.$$

Then $g(n)$ (asymptotically) dominates $T(n)$, and $T(n)$ is (asymptotically) dominated by $g(n)$.

The set of all functions that $g(n)$ dominates is denoted by $\boldsymbol{O(g(n))}$.

If $g(n)$ dominates $T(n)$, then $T(n)$ is $O(g(n))$, meaning $T(n)$ is in the set $O(g(n))$.

If a function $g(n)$ dominates a time complexity function $T(n)$ and $T(n)$ also dominates $g(n)$, then $T(n)$ has **order** $\boldsymbol{O(g(n))}$, and $T(n)$ and $g(n)$ have the **same order of magnitude**.

4 The **order hierarchy** of common big-oh sets is

$$\underset{\text{fast}}{O(1)} \subset O(\log_2 n) \subset O(n) \subset O(n\log_2 n) \subset O(n^2) \subset \ldots \subset O(n^k) \subset \ldots \subset O(2^n) \subset \underset{\text{slow}}{O(n!)}.$$

The time complexity function $T(n)$ of an algorithm is **placed** in the set $O(g(n))$ in the hierarchy if $T(n)$ has order $O(g(n))$. The further to the left in the hierarchy $T(n)$ is placed, the faster the algorithm (for large enough n).

To find the order of a given time complexity function $T(n)$ consisting of the sum of one or more terms:

- take the base of each logarithm to be 2;
- take each coefficient to be 1;
- take each constant term to be 1.

Then the term $g(n)$ whose set lies furthest to the right in the order hierarchy is the dominant term, and $T(n)$ has order $O(g(n))$.

Section 2 Stacks and lists

1 The **stack data type** consists of data stored in an ordered set (stack) of adjacent storage cells together with the basic operations:

TOP(s) = top item in stack s;

DEPTH(s) = number of items in stack s;

PUSH(item, s) = stack created by pushing (adding) item onto top of stack s;

POP(s) = stack created by popping (removing) top item from stack s;

ISEMPTYSTACK? = TRUE if s is the empty stack, and FALSE otherwise.

2 The **list data type** is a data store k in which each item keeps an address that **points** to the next item, together with the basic operations:

FIRST(k) = first item in list k;

ITEM(i, k) = ith item in list k;

LENGTH(k) = number of items in list k;

INSERT(item, i, k) = insert item after $(i-1)$th item in list k, so that it becomes the ith item of the new list formed;

DELETE(i, k) = remove ith item from list k.

3 A **depth-first search** of a rooted path graph starts at the root vertex and moves from level to level, comparing the item at each vertex with the given item.

Theorem 2.1

The time complexity function for a depth-first search of a rooted path graph is of order $O(n)$.

Algorithm: SEARCH 1 **for a given item in a list** $\boldsymbol{k}$ **of length** $\boldsymbol{n}$

```
STEP 1 Set the value of a variable $i = 0$.
STEP 2 Increase the value of $i$ by 1 and compare ITEM($i$, $k$)
       with the given item. If they are the same, or if they
       are not the same but $i = n$, STOP. Otherwise repeat
       Step 2.
```

Algorithm: SEARCH 2 **for a given item in a list** $\boldsymbol{k}$

```
STEP 1 If LENGTH($k$) = 0, the list is empty, so STOP.
STEP 2 Compare the given item with ITEM(1, $k$). If they are
       the same, STOP.
STEP 3 Use SEARCH 2 on the list DELETE(1, $k$).
```

4 A **recurrence system** is a system of equations of the form

$$T(k) = a(n)$$
$$T(n) = b(n) + T(n-1) \qquad (n > k).$$

Solution by iteration is the derivation of a formula for $T(n)$ using a recurrence system.

5

Theorem 2.2

The time complexity function for a search for all repeated items in a rooted path graph is of order $O(n^2)$.

6 The **bubble sort** algorithm takes the form of a number of 'passes' over all the items in a list. For the first pass, start at the first item in the list, at the root vertex, and compare it with the item at the second vertex. If the first item is 'larger', exchange the two vertices; if the first item is not larger, do nothing. Work along the graph, comparing the ith vertex with the $(i+1)$th vertex, for each i, and exchanging them if necessary. The number of such passes made is $n - 1$, each pass using one fewer comparison than the previous one.

Theorem 2.3

The time complexity function for a bubble sort is of order $O(n^2)$.

7 **Algorithm to merge two sorted lists**

```
STEP 1 Start a new list, with no items in it.
STEP 2 Compare the first items in both sorted lists, delete
       the smaller of these from its list, and insert it at the
       end of the new list. Repeat until one list is empty.
STEP 3 Insert what remains of the non-empty list at the
       end of the new list, and STOP.
```

Theorem 2.4

The time complexity function for merging two sorted lists of lengths m and n, where $m \leq n$, is of order $O(n)$.

8 The **merge sort** algorithm first bisects a list into two lists of equal (or nearly equal) length. It then bisects each such half-length list into two equal (or nearly equal) quarter-length lists. The bisection process continues until a number of pairs of short lists are obtained, each with one

or two items. These short lists are easily sorted, as there is nothing to do for a one-item list, and a single comparison to make for a two-item list. The pairs of short lists are now merged to give pairs of lists, each with three or four items. These pairs are merged, and so on, until the whole list is sorted.

Theorem 2.5

The time complexity function for a merge sort on a list of length n is of order $O(n\log_2 n)$.

9 The **divide-and-conquer method** of problem solving breaks down a problem into subproblems, solves these subproblems, and then combines the solutions to the subproblems into a solution for the original problem. The method is *recursive*; the subproblems themselves can be solved using the divide-and-conquer method.

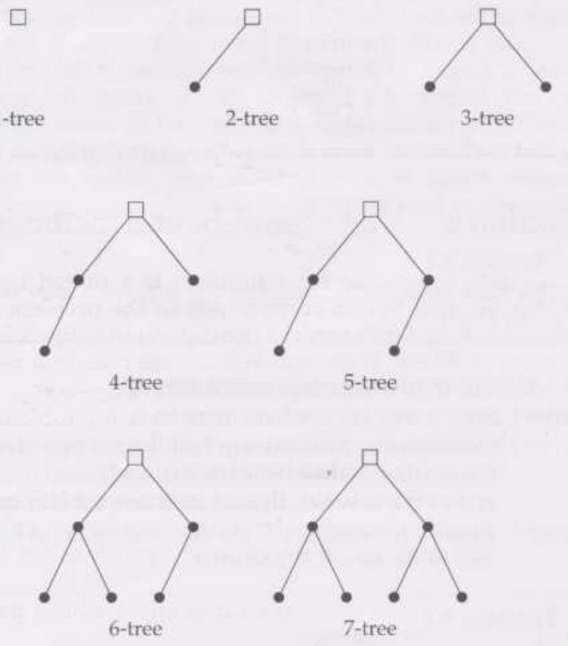
Section 3 Binary trees

1 For each vertex v of a binary tree, an adjacent vertex below and to the left can be regarded as the root of another binary tree, the **left subtree** of v, and similarly an adjacent vertex below and to the right is the root of the **right subtree** of v.

2 Algorithm: GROW-TREE for constructing a binary tree from a list

```
STEP 1  If the list is empty, STOP.
STEP 2  Find the centre or bicentre of the graph of the list.
        If it has a centre, choose this as the root of the
        binary tree; if it has a bicentre, choose the
        rightmost vertex of the two as the root.
STEP 3  The root divides the list into a left list and a right
        list (one or both of which may be empty). Apply
        GROW-TREE to both of these to obtain the left and
        right subtrees.
```

The use of the GROW-TREE algorithm on lists of lengths 1, …, 7 yields the following catalogue of binary trees.



3 The **height** of a binary tree is the number of vertices in a path of maximum length, starting from the root.

A binary tree is **balanced** if, at each vertex v, the heights of the left subtree of v and the right subtree of v do not differ by more than 1.

4 The **binary tree data type** is a data store T, corresponding to a binary tree, in which each item keeps two addresses that point to its two subtrees, together with the basic operations:

ROOT(T) = item at root vertex;
LEFT(T) = left subtree of T;
RIGHT(T) = right subtree of T;
MAKE-TREE(T_1, item, T_2) = tree constructed with the item as its root, T_1 as its left subtree and T_2 as its right subtree;
ISEMPTYTREE? = TRUE if T is the empty tree, and FALSE otherwise.

5 Given a set of items that can be ordered, a **binary search tree** is a binary tree in which each vertex v represents one of these items in such a way that

- v is larger than each item below and to the left of v;
- v is smaller than each item below and to the right of v.

Algorithm: SEARCH a binary search tree

```
STEP 1  Compare the search item with the ROOT. If they are
        the same, STOP.
STEP 2  If the search item is smaller than the ROOT, then
        apply SEARCH to the LEFT tree, if it exists. Otherwise
        STOP.
        If the search item is larger than the ROOT, then
        apply SEARCH to the RIGHT tree, if it exists.
        Otherwise STOP.
```

Theorem 3.1

The time complexity function for a search on a binary search tree of height h is of order $O(h)$.

The time complexity function for a search on a balanced binary search tree with n vertices is of order $O(\log_2 n)$.

6 The idea of **depth-first search** of a graph is to penetrate as deeply as possible into the graph before fanning out to the other vertices.

Algorithm: DEPTH-FIRST SEARCH on a binary tree

```
STEP 1  Start at the ROOT. Add it as the rightmost vertex of
        the list. If there are no subtrees, STOP.
STEP 2  If there is a LEFT subtree, then apply DEPTH-FIRST
        SEARCH to the LEFT subtree.
        If there is a RIGHT subtree, then apply DEPTH-FIRST
        SEARCH to the RIGHT subtree.
```

Algorithm for manipulating the stack in a depth-first search

```
STEP 1  PUSH the root vertex onto the empty stack.
STEP 2  If the top vertex of the stack is adjacent to a new
        vertex, PUSH this vertex onto the stack. Otherwise,
        POP the top vertex off the stack.
STEP 3  If the stack is empty, STOP. Otherwise, return to
        Step 2.
```

This algorithm can be applied to any rooted tree, not just to a binary tree. The algorithm can be used to search any connected graph: the only change is that the 'root vertex' in Step 1 becomes the 'start vertex' — any vertex can be chosen as the start vertex.

The tree consisting of the vertices visited and the edges traversed in a depth-first search on a connected graph is called a **depth-first search spanning tree**.

7 A **queue** is a data store similar to a stack; the difference is that a queue is FIFO (First In, First Out) whereas a stack is LIFO (Last In, First Out).

8 The idea of **breadth-first search** of a graph is to fan out to as many vertices as possible before penetrating deep into the graph. This means that all the vertices adjacent to the current vertex must be visited before going on to another vertex.

Algorithm for manipulating the queue in a breadth-first search

```
STEP 1 INSERT the root vertex into the empty queue.
STEP 2 If the first vertex of the queue is adjacent to a new
       vertex, INSERT this vertex at the end of the queue.
       Otherwise, DELETE the first vertex of the queue.
STEP 3 If the queue is empty, STOP. Otherwise, return to
       Step 2.
```

The tree consisting of the vertices visited and edges traversed in a breadth-first search on a connected graph is called a **breadth-first search spanning tree**.

9

Theorem 3.2

The time complexity functions for depth-first search and breadth-first search of a connected graph with m edges are both of order $O(m)$.

Section 4 Quad trees

1 A ***k*-screen** consists of $2^{k-1} \times 2^{k-1}$ *pixels* (small squares).

A **quad tree** that represents a k-screen is a rooted tree with at most k levels. Each end-vertex has 1 or 0 stored at it, denoting that the pixel (or quadrant of pixels) it represents is black or white, respectively. Each other vertex has a set of four vertices below it, ordered from left to right, this ordering being denoted by the labels A, B, C, D. At each level, each set of four vertices corresponds to four quadrants of the screen — the A-vertex to the top-right quadrant, the B-vertex to the top-left quadrant, and so on, in an anticlockwise direction.

2 To **rotate** an image on a k-screen anticlockwise through a right angle, rotate it about its centre. This rotation is represented by a cycle of the quadrants

$$A \to B \to C \to D \to A.$$

Other rotations can be represented similarly.

To **reflect** an image in a horizontal line through the middle of a k-screen, interchange the quadrants A and D and the quadrants B and C, according to the scheme

$$A \leftrightarrow D \quad \text{and} \quad B \leftrightarrow C.$$

Other reflections can be represented similarly.

To **magnify** an image in one of the principal quadrants of a k-screen by a factor 2, interpret the subtree for that quadrant as the quad tree for the whole screen.

Magnification by other powers of 2 is also possible by moving further down the levels of the quad tree and detaching the subtrees.

To reduce an image on a k-screen by factors of 2, move all vertices down one or more levels.

3 The pixel immediately above another pixel is called the **north neighbour** of the pixel.

For (blocks of) pixels, the B-quadrant and C-quadrant are **opposites**, and the A-quadrant and D-quadrant are opposites, as far as being north neighbours is concerned.

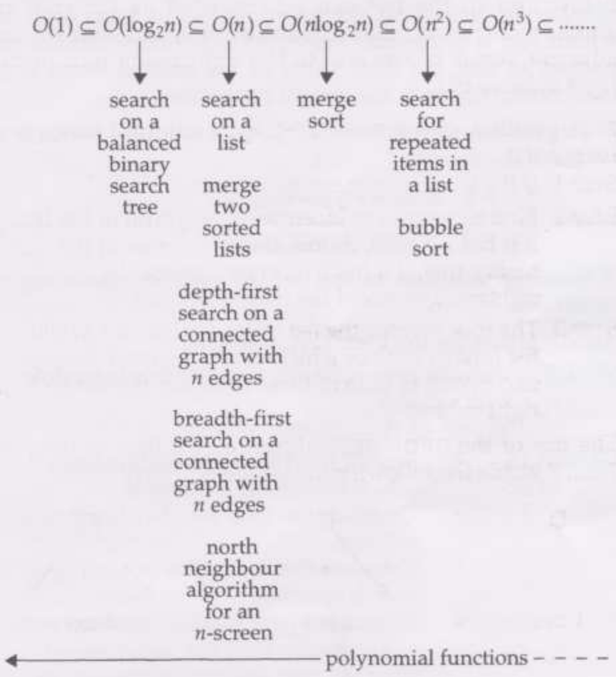
Algorithm for finding the north neighbour of a vertex of a quad tree

```
STEP 1 Set up an empty 'path' stack to keep track of the
       vertices that are passed. Make the start vertex the
       'current' vertex.
STEP 2 If the 'current' vertex is the root vertex, and a
       lower (C or D) vertex is not on the TOP of the 'path'
       stack, then no north neighbour exists, so STOP.
       Otherwise, PUSH the 'current' vertex label onto the
       'path' stack. Move up to the parent vertex and
       make it the 'current' vertex. If the TOP of the 'path'
       stack is not a lower (C or D) vertex, repeat Step 2.
STEP 3 POP a vertex label from the 'path' stack. Move
       down to the vertex with the opposite label to the
       one just taken off the stack, and make it the
       'current' vertex.
       Repeat Step 3 until the 'path' stack is empty. The
       'current' vertex is then the north neighbour of the
       start vertex.
```

The north neighbour algorithm can be adapted to yield (the colours of) the other seven neighbouring pixels — the south, east, west, north-east, north-west, south-east and south-west neighbour pixels.

The algorithms for finding one, or all, of the eight neighbours of a given pixel are all of order $O(k)$.

Time complexity functions for computer algorithms on n items of data



O(1) ⊆ O(log₂n) ⊆ O(n) ⊆ O(nlog₂n) ⊆ O(n²) ⊆ O(n³) ⊆ ......
search on a balanced binary search tree
search on a list
merge two sorted lists
depth-first search on a connected graph with n edges
breadth-first search on a connected graph with n edges
north neighbour algorithm for an n-screen
merge sort
search for repeated items in a list
bubble sort
polynomial functions - - - -

Section 5 Branch-and-bound methods

1 A **state space tree** for a problem is a rooted tree in which the root vertex corresponds to the problem as a whole and the other vertices correspond to subproblems. The subproblems at the end-vertices are ones that can be solved immediately. Each parent vertex is:

either an **AND vertex**, in which case its (sub)problem can be solved immediately once *all* the subproblems at its children vertices have been solved;

or an **OR vertex**, in which case its (sub)problem can be solved immediately once *any one* of the subproblems at its children vertices has been solved.

An **AND tree** is a state space tree all of whose parent vertices are AND vertices.

An **OR tree** is a state space tree all of whose parent vertices are OR vertices.

And **AND/OR tree** is a state space tree that has a mixture of parent vertex types.

2 Branch-and-bound method

Given a problem that requires the optimization of some objective, determine an appropriate **bound** for the objective. Break the problem down into a set of subproblems and determine the bound for each.

For each subproblem that is not immediately solved or cannot immediately be shown to have a non-optimal or no solution, break it down further into subproblems and determine the corresponding bounds. Repeat this process until each subproblem is solved or has been shown to have a non-optimal or no solution.
An optimal solution to the original problem is given by any subproblem with a 'best' bound.

3 The divide-and-conquer method corresponds to an AND tree, since *every* subproblem has to be solved in order to solve the initial problem. The branch-and-bound tree is an OR tree, so that the solution to each subproblem is a solution to the original problem.

4 Knapsack problem

A hiker is planning a journey, but has a knapsack that can accommodate only a certain total weight. There are a number of items that the hiker wishes to take, each of which has a particular value. Which items should be packed so that the total value of the packed items is a maximum, subject to the weight restriction?

For a problem with n items $x_1, \ldots, x_n$, a **solution vector** is a vector of the form $(x_1, \ldots, x_n)$,

$$\text{where } \begin{cases} x_i = 1 & \text{if item } i \text{ is packed;} \\ x_i = 0 & \text{if item } i \text{ is not packed.} \end{cases}$$

A **feasible solution** is one which satisfies the weight constraint.

An algorithm for the knapsack problem

```
START      with the zero vector (0, 0, ..., 0); this is feasible
           with value 0.
           STORE (0, 0, ..., 0) and value 0.
GENERAL    •  Branch from the first solution from
   STEP       which branching is possible.
           •  Calculate the total weight of each new
              solution.
           •  Calculate the total value of each new
              feasible solution.
              If there is a new feasible solution with
              value greater than the value stored,
              store the new solution vector and its
              value instead.
           •  Mark a vertex with a □ if it corresponds
              to:
              □  a vector which equals or exceeds
                 the weight restriction;
              □  a vector which ends in 1.
REPEAT     the GENERAL STEP until no more branching
           is possible.
STOP       The stored solution vector and its value gives
           an optimum solution.
```

5 An algorithm for the travelling salesman problem

```
START      with a given $n \times n$ table of distances,
           corresponding to a complete weighted graph
           with $n$ vertices.
           Carry out the initial row and column
           reduction, and calculate the initial lower bound
           for the total length of the route.
GENERAL    •  Consider all the edges with zero
   STEP       weight, and choose an allowable
              edge $e$ whose exclusion leads to the
              greatest increase in the lower bound;
              if there are several such edges, choose
              the first.
              (Avoid cycles with fewer than $n$ edges.)
           •  Consider the consequences of including
              $e$ and excluding $e$. Use row and column
              reduction to determine these
              consequences, in terms of increases in
              the lower bound. Choose the option
              which gives the smaller lower bound; if
              the lower bounds are equal, include the
              edge $e$.
              STORE the current list of included
              edges, and the current lower bound.
           •  Continue from the current position
              unless the chosen option has a lower
              bound greater than a previously
              eliminated option, in which case
              backtrack to the earlier position.
REPEAT     the GENERAL STEP until a cycle with $n$
           edges has been created.
STOP       The stored list of edges gives an optimum
           solution.
```

Networks 4 Physical networks

Section 1 Modelling physical networks

1 Through and across variables

A physical network can be considered to be a number of individual components connected together.

In this unit it is assumed that the behaviour of a component does not change if it is moved from one system to another, and that the behaviour is time-invariant and linear.

The point at which one component is connected to another is called a **terminal**.

An **across variable** represents the difference in a quantity between one terminal of a component and another. A **through variable** represents the quantity flowing through a component.

2 Equations relating through and across variables

Kirchhoff's current law

The current flow into any vertex of an electrical network is equal to the current flow out of it.

Kirchhoff's voltage law

The algebraic sum of the potential differences across all the components around any circuit (cycle) in an electrical network is zero.

Analogues of these laws can be given for many types of network containing through and across variables. The analogue of Kirchhoff's current law for a network carrying a flow is the *flow conservation law*: what flows into a vertex of a network (other than a source or sink) must flow out of it. The analogue of Kirchhoff's voltage law is the *potential difference conservation law*: the sum of the potential differences around any circuit of a network is zero.

3 Graphical representation of components

A 2-terminal component can be represented by two vertices joined by a single edge.

Associated with the edge are two **edge variables** — one through variable and one across variable.

A *reference direction* for the edge variables is obtained by assigning an arbitrary direction to each edge, indicated by drawing an arrow alongside the edge. Currents which actually flow through the component corresponding to the edge in the direction of the arrow are regarded as positive, and currents in the opposite direction are regarded as negative. Similarly, voltage differences between the terminals corresponding to two vertices are always taken as the voltage at the tail of the arrow minus the voltage at the head. Thus if the voltage at the tail is greater than the voltage at the head, the voltage difference is positive: if the voltage at the tail is less, then the voltage difference is negative. The reference direction is purely arbitrary and does not necessarily represent the actual direction of the current flow or voltage difference.

Any system consisting entirely of 2-terminal components can be represented by a graph whose edges correspond to the components and whose vertices correspond to the terminals. Such a graph, with a reference direction associated with each edge, is called an **oriented graph**.

4 Two-terminal components

An equation relating the two edge variables for a 2-terminal component is a **component equation**.

A **resistor** is modelled by a 2-terminal component whose edge variables v and i satisfy a relationship of the form

$$v = Ri,$$

where R is a constant, usually called the **resistance**.

This relationship is known as **Ohm's law**. It is sometimes written in the form $i = G\,v$, where $G\ (= 1/R)$ is the **conductance** of the resistor.

A **capacitor** is modelled by a 2-terminal component whose edge variables v and i satisfy a relationship of the form

$$i = C\frac{dv}{dt},$$

where C is a constant, usually called the **capacitance**.

An **inductor** is modelled by a 2-terminal component whose edge variables v and i satisfy a relationship of the form

$$v = L\frac{di}{dt},$$

where L is a constant, usually called the **inductance**.

Components such as batteries and electric generators are modelled by the concept of an **ideal independent source**. An ideal independent *voltage* source maintains a *prescribed voltage* across its terminals, irrespective of the size of the current flowing between them. An ideal independent *current* source maintains a *prescribed current*, irrespective of the value of the voltage across its terminals.

Some important types of component, together with their graphical representations and their corresponding component equations are listed on page 42.

5 Multi-terminal components

A **multi-terminal component** is a component which has more than two terminals, and which cannot therefore be represented by a single edge joining two vertices. Two important types of electrical multi-terminal component are *transformers* and *transistors*.

Graphical representations and component equations of some multi-terminal components are given on page 43.

A 3-terminal component is represented by *two* edges and has *two* component equations.

Similarly, any labelled tree with n vertices is a suitable representation of an n-terminal component. It follows that *any n-terminal component can be regarded as $n - 1$ 2-terminal components*, one corresponding to each edge of the tree.

6 Kirchhoff's laws (alternative forms)

Kirchhoff's laws can be formulated in graph-theoretical terms as follows.

Vertex law

The algebraic sum of the through variables associated with the edges at each vertex of an oriented graph is zero.

Here 'the algebraic sum' means that the orientation of the edges must be taken into account, so if edges oriented *into* a vertex are counted as *positive*, then edges oriented *out of* a vertex are counted as *negative*.

Cycle law

The algebraic sum of the across variables associated with the edges in any cycle of an oriented graph is zero.

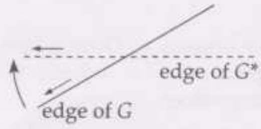
Here 'the algebraic sum' means that a particular direction around a cycle (say clockwise) is chosen, and edges oriented in the *same direction* are considered as *positive* and edges oriented in the *opposite direction* as *negative*.

7 Dual electrical networks

The idea of *duality* can be applied to (planar) electrical networks as follows.

Given an electrical network N, draw a plane drawing of a corresponding oriented graph, form the dual of this planar graph as described in *Graphs* 3, and then construct the corresponding network N^*, called the **dual network**.

The rule for associating an arrow with each edge of the dual graph is illustrated below.



go *clockwise* from edge of G to edge of G^* in *same* direction

Voltage in the original network N corresponds to *current* in the dual network N^* and *vice versa;* the roles of v and i in the component equations are interchanged. The following table shows correspondences between the components in the two networks N and N^*.

network N	dual network N^*
voltage	current
current	voltage
resistor (resistance R)	resistor (resistance $1/R$)
capacitor (capacitance C)	inductor (inductance C)
inductor (inductance L)	capacitor (capacitance L)

Section 2 Electrical networks: matrix equations

1 Linearly independent equations

A number of equations are **linearly dependent** if at least one of them 'depends' on the others, in the sense that it can be obtained by adding or subtracting multiples of the others. If none of the equations depends on the others in this way, then the equations are **linearly independent**.

2 Fundamental cycles

A number of cycles in a graph are **linearly independent** if the corresponding voltage equations are linearly independent.

Let G be a connected graph. A **spanning tree** in G is a subgraph of G which includes all the vertices of G and is also a tree. The edges of the tree are called **branches**. If all the edges of a spanning tree are removed from G, the remaining subgraph is called a **co-tree** and its edges are called **chords**.

Let G be a connected graph, and let T be a spanning tree of G. The set of **fundamental cycles** associated with T consists of the cycles of G each of which is obtained by adding a single chord to T. The corresponding voltage law equations are called the **fundamental cycle equations**.

If G has n vertices and m edges, then any spanning tree gives rise to $m - n + 1$ chords, and so there are $m - n + 1$ cycles in a fundamental set. This number is the largest number of linearly independent cycles in G, and is called the **cycle rank** of G.

In any electrical network problem, all the information contained in Kirchhoff's voltage law can be obtained by:

(a) choosing a spanning tree;

(b) finding the fundamental cycles associated with it;

(c) finding the corresponding fundamental cycle equations.

The fundamental cycle equations can be represented in matrix form. The rows of the matrix correspond to the chords associated with a spanning tree, and the columns correspond to all the edges of the graph G. (It is usual to write the columns corresponding to the branches of the spanning tree first, followed by those for the chords.) If G has n vertices and m edges, an $(m - n + 1) \times m$ matrix is obtained. This matrix is called the **fundamental cycle matrix**, and is denoted by $\mathbf{C}_f$.

To construct this matrix, add each chord to the spanning tree in turn, and look at the edges in the corresponding fundamental cycle. Trace around this cycle in the direction of the chord, then fill in the row corresponding to the given chord, writing:

1 in each column corresponding to an edge of the cycle oriented in the same direction as the chord;

–1 in each column corresponding to an edge oriented in the opposite direction;

0 in each column corresponding to an edge which is not in the cycle.

The fundamental cycle equations are written in matrix form as

$$\mathbf{C}_f\, \mathbf{v} = \mathbf{0},$$

where $\mathbf{v}$ is the column vector of edge voltages, written in the same order as the columns of the matrix, and $\mathbf{0}$ is the appropriate zero vector.

3 Fundamental cutsets

Let G be a connected graph, and let T be a spanning tree of G. The **set of fundamental cutsets** associated with T consists of the cutsets of G obtained by removing a branch of T, thus separating the tree into two parts X and Y, and listing the edges of G joining a vertex in X and a vertex in Y. The corresponding current law equations are called the **fundamental cutset equations**.

A number of cutsets in a graph G are **linearly independent** if the corresponding current law equations are linearly independent. Thus the cutsets in a set of fundamental cutsets are linearly independent, since each cutset contains one edge (the branch of the spanning tree) not contained in any of the other cutsets. If G has n vertices, then any spanning tree has $n - 1$ branches, and so there are $n - 1$ cutsets in a fundamental set. This number is the largest number of independent cutsets in G, and is called the **cutset rank** of G.

In any electrical network problem, all the information contained in Kirchhoff's current law can be obtained by:

(a) choosing a spanning tree;

(b) finding the fundamental cutsets associated with it;

(c) finding the corresponding fundamental cutset equations.

The fundamental cutset equations can be represented in matrix form. The rows of the matrix correspond to the branches of a spanning tree, and the columns correspond to all the edges of the graph G. (It is usual to write the columns corresponding to the branches first, followed by those for the chords.) If G has n vertices and m edges, an $(n - 1) \times m$ matrix is obtained. This matrix is called the **fundamental cutset matrix**, and is denoted by $\mathbf{D}_f$.

To construct this matrix, take each branch of the spanning tree in turn, and look at the edges in the corresponding fundamental cutset. Then fill in the row corresponding to the given branch, writing:

1 in each column corresponding to an edge of the cutset oriented from X to Y;

–1 in each column corresponding to an edge of the cutset oriented from Y to X;

0 in each column corresponding to an edge not in the cutset.

The fundamental cutset equations are written in matrix form as

$$\mathbf{D}_f\, \mathbf{i} = \mathbf{0},$$

where $\mathbf{i}$ is the column vector of edge currents, written in the same order as the columns of the matrix, and $\mathbf{0}$ is the appropriate zero vector.

4 Obtaining the fundamental cycle and cutset matrices

Let G be an oriented graph with n vertices and m edges. The **incidence matrix B(G)** is the $n \times m$ matrix in which the entry in row i and column j is

1	if edge j is incident with, and oriented away from, vertex i;
–1	if edge j is incident with, and oriented towards, vertex i;
0	if edge j is not incident with vertex i.

Any single row of $\mathbf{B}(G)$ can be omitted without loss of information. The vertex corresponding to the row omitted is the **reference vertex** and the resulting matrix is the **reduced incidence matrix**, denoted by $\mathbf{B}_0$.

The matrix $\mathbf{B}_0$ is partitioned into two parts, one part corresponding to the branches, and one part corresponding to the chords. This partitioned matrix can be written as $\mathbf{B}_0 = [\mathbf{B}_t \mid \mathbf{B}_c]$, where $\mathbf{B}_t$ is the tree part and $\mathbf{B}_c$ is the co-tree part.

Theorem 2.1

The fundamental cycle matrix $\mathbf{C}_f$ and the fundamental cutset matrix $\mathbf{D}_f$ can be expressed in terms of $\mathbf{B}_t$ and $\mathbf{B}_c$ as follows:

fundamental cycle matrix $\quad \mathbf{C}_f = \left[-(\mathbf{B}_t^{-1}\mathbf{B}_c)^T \,\middle|\, \mathbf{I}_{m-n+1}\right]$

fundamental cutset matrix $\quad \mathbf{D}_f = \left[\mathbf{I}_{n-1} \,\middle|\, \mathbf{B}_t^{-1}\mathbf{B}_c\right].$

5 Tellegen's theorem

Theorem 2.2: Tellegen's theorem

Suppose that there are two electrical networks which can be represented by the same oriented graph with n vertices and m edges. Then

$$\sum_{k=1}^{m} v_k i_k = v_1 i_1 + v_2 i_2 + \cdots + v_m i_m = 0,$$

where v_k is the voltage associated with the kth edge for the first network, and i_k is the current associated with the kth edge for the second network.

Section 3 Electrical networks: solving the network equations

1 Combining the matrix equations

There are three matrix equations, corresponding to the component equations, the fundamental cycles, and the fundamental cutsets.

When the voltages and currents in each equation are written in the same order, the three matrix equations can be combined into one as follows:

$$\mathbf{T}\left[\begin{array}{c}\mathbf{v}\\ \hdashline \mathbf{i}\end{array}\right] = \mathbf{f}, \quad \text{(component equation)}$$

$$\left[\mathbf{C}_f \,\vdots\, \mathbf{0}\right]\left[\begin{array}{c}\mathbf{v}\\ \hdashline \mathbf{i}\end{array}\right] = \mathbf{0}, \quad \text{(fundamental cycle equation)}$$

$$\left[\mathbf{0} \,\vdots\, \mathbf{D}_f\right]\left[\begin{array}{c}\mathbf{v}\\ \hdashline \mathbf{i}\end{array}\right] = \mathbf{0}, \quad \text{(fundamental cutset equation)}$$

which can be written as one equation

$$\left[\begin{array}{c:c}\multicolumn{2}{c}{\mathbf{T}}\\ \hdashline \mathbf{C}_f & \mathbf{0}\\ \hdashline \mathbf{0} & \mathbf{D}_f\end{array}\right]\left[\begin{array}{c}\mathbf{v}\\ \hdashline \mathbf{i}\end{array}\right] = \left[\begin{array}{c}\mathbf{f}\\ \hdashline \mathbf{0}\\ \hdashline \mathbf{0}\end{array}\right]$$

or

$$\mathbf{Hx} = \mathbf{y},$$

where

$\mathbf{H}$	is a $2m \times 2m$ matrix containing component, cycle and cutset information;
$\mathbf{x}$	is a column vector containing all the voltages and currents;
$\mathbf{y}$	is a column vector consisting mainly of zeros, but also including terms such as I and $-V$ (source functions).

2 Solving the matrix equation $\mathbf{Hx} = \mathbf{y}$

Gaussian elimination is a systematic way of solving a linearly independent system of n simultaneous linear equations in n variables by eliminating the variables one at a time. This is done by performing certain allowable operations on the rows of the matrix of coefficients $\mathbf{H}$ so as to reduce all the elements below the main diagonal (the diagonal from top left to bottom right) to zero. The equations $\mathbf{Hx} = \mathbf{y}$ can then be successively solved, starting with the last equation, until all the required variables are obtained.

The operations allowed on the rows of the matrix are:

- interchange two rows;
- multiply any row by a non-zero number;
- add any row to any other row.

If these row operations are also carried out on the column vector $\mathbf{y}$, then each of them corresponds to an operation on the corresponding equations which leaves the solution of the equations unchanged.

3 State equations

For networks involving capacitors and inductors, the component equations involve not only v and i, but also their derivatives. The matrix equation can be adapted to take the derivatives into account.

The matrix equation can be written in the form

$$\mathbf{Hx} = \mathbf{y} + \mathbf{K}\dot{\mathbf{x}},$$

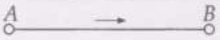
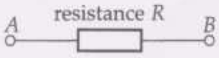
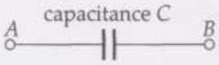
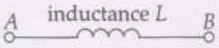
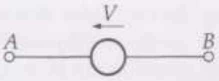
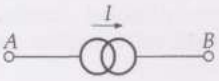
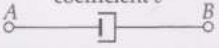
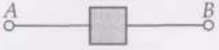
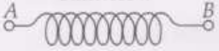
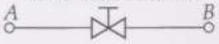
where

$\mathbf{x}$	is a column vector involving all the voltages and currents;
$\dot{\mathbf{x}}$	is a column vector involving their derivatives;
$\mathbf{y}$	is the 'source function vector' introduced earlier;
$\mathbf{H}$	is the matrix involving the components and the fundamental cycles and cutsets;
$\mathbf{K}$	is a matrix isolating the derivatives of the edge variables.

The variables whose derivatives occur in the component equations appear *last of all* in the column vectors $\mathbf{x}$ and $\dot{\mathbf{x}}$. These variables are called **state variables**. The values of the state variables represent the *state* of the system.

The **state equations** are first-order differential equations that relate the derivatives of the state variables to the state variables and the independent sources.

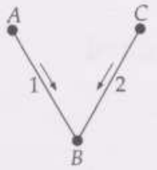
Graphical representation of 2-terminal components

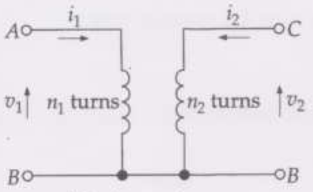
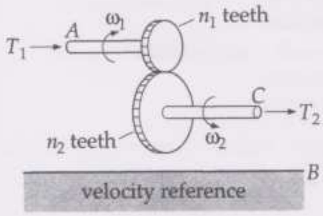
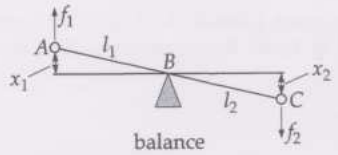
	graphical representation		
two–terminal component	through variable	across variable	component equation
resistance R  resistor	current i	voltage v	$v = Ri$
capacitance C  capacitor	current i	voltage v	$i = C\dfrac{dv}{dt}$
inductance L  inductor	current i	voltage v	$v = L\dfrac{di}{dt}$
 voltage source	current i	voltage v	$v = V$
 current source	current i	voltage v	$i = I$
damping coefficient c  damper	force f	velocity u	$f = cu$
mass m  mass	force f	velocity u	$f = m\dfrac{du}{dt}$
stiffness k  spring	force f	velocity u	$u = \dfrac{1}{k}\dfrac{df}{dt}$
valve 'resistance' R  hydraulic valve	flow q	pressure p	$p = Rq$

(The relationship $f = m\frac{du}{dt}$ for a spring is often written in the form $f = kx$, known as Hooke's law, where x is the extension of the spring due to an applied force f.)

Graphical representation of 3-terminal components

graphical representation



three–terminal component	through variables	across variables	component equations
 A i_1 i_2 C v_1 n_1 turns n_2 turns v_2 B B ideal transformer	currents i_1 and i_2	voltages v_1 and v_2	$v_2 = \frac{n_2}{n_1} v_1$ $i_2 = -\frac{n_1}{n_2} i_1$
 T_1 A ω_1 n_1 teeth C T_2 n_2 teeth ω_2 B velocity reference gear train (friction and inertia neglected)	torques T_1 and T_2	angular velocities ω_1 and ω_2	$T_2 = \frac{n_2}{n_1} T_1$ $\omega_2 = -\frac{n_1}{n_2} \omega_1$
 f_1 A l_1 B x_1 x_2 l_2 C f_2 balance	forces f_1 and f_2	displacements x_1 and x_2	$f_2 = \frac{l_1}{l_2} f_1$ $x_2 = -\frac{l_2}{l_1} x_1$

Design 4 Block designs

Section 1 Blocking in designed experiments

1 Varieties, plots and blocks

In an experiment, the **varieties** are the items being compared.

A **plot** is a unit of experimental material on which a single variety is tested.

A **block** is a group of plots which are similar in the sense that, if the same variety is tested on each, then similar results can be expected.

2 Incomplete block designs

An **incomplete block design** consists of:

- a set of **varieties**;
- a set of **blocks**, each containing the same number of varieties;
- a list showing the varieties allocated to the plots in each block.

No block contains every variety, and no block contains any variety more than once.

Notation

v denotes the number of varieties;

b denotes the number of blocks;

k denotes the **block size** — the number of plots in each block.

A block design is **equi-replicate** if each variety occurs the same number of times; this number is the **replication**, and is denoted by r.

In this unit, all incomplete block designs are assumed to be equi-replicate, with the **parameters**:

- v varieties
- b blocks
- k varieties in each block, where $k < v$
- r occurrences of each variety.

The **v**ariety letters are **v** and **r**, and the **b**lock letters are **b** and **k**.

Theorem 1.1

In an equi-replicate incomplete block design, the parameters v, r, b and k are related by the equation

$$vr = bk.$$

3 Isomorphic designs

Two block designs are **isomorphic** if one can be obtained from the other by relabelling the blocks and/or the varieties.

4 Connected designs

Two varieties are **directly comparable** if there is a block that contains both.

Two varieties A and B are **indirectly comparable** if there is a chain of varieties $A_1, \ldots, A_n$, such that A and A_1, A_1 and $A_2, \ldots$, and A_n and B are all directly comparable.

A block design is **connected** if each pair of varieties is directly or indirectly comparable.

5 Incidence matrices

The **incidence matrix B** of a block design with v varieties and b blocks is the $v \times b$ matrix with rows labelled by the varieties and columns labelled by the blocks, in which the entry in row A and column $\mathbb{B}$ is

1 if variety A lies in block $\mathbb{B}$;

0 if variety A does not lie in block $\mathbb{B}$.

6 Constructing block designs

All-combinations method

For the blocks, take all possible combinations of k varieties chosen from the set of v varieties.

Circle construction

Place the letters representing the varieties around a circle, and mark a starting point;
for the first block, take the first k varieties, proceeding clockwise from the starting point;
for the second block, take the next k varieties, still proceeding clockwise;
and so on.

When the varieties are letters, rather than numbers, the construction is similar.

Cyclic construction

Denote the varieties by $0, 1, \ldots, v-1$;
for the first block, choose any k of these integers;
for the second block, add 1 to each variety in the first block, replacing v by 0 if it occurs;
for the third block, add 1 to each variety in the second block, replacing v by 0 if it occurs;
and so on.

When the varieties are letters, rather than numbers, the construction is similar.

7 New designs from old

Complement of a design

From a given design Δ, the **complement** $\overline{\Delta}$ is constructed as follows.

For each block $\mathbb{B}$ of Δ, construct a block $\mathbb{B}$ of $\overline{\Delta}$ by taking the varieties in block $\mathbb{B}$ of $\overline{\Delta}$ to be those varieties that do not occur in block $\mathbb{B}$ of Δ.

The complement of $\overline{\Delta}$ is Δ.

Dual of a design

From a given design Δ, the **dual** Δ^* is constructed as follows.

Interchange the roles of varieties and blocks, re-interpreting 'contains' as 'occurs in', and *vice versa.*

The dual of Δ^* is Δ.

Section 2 Balanced designs

1 Concurrence

Let A and B be two distinct varieties in an incomplete block design. The **concurrence** of A and B is the number of blocks where both A and B occur, and is denoted by λ_{AB}.

An incomplete block design Δ is **balanced** if there is a constant number λ such that each pair of distinct varieties of Δ occur together in exactly λ blocks.

The number λ is the **concurrence** of the balanced design Δ.

The **concurrence matrix C** of an incomplete block design is the $v \times v$ matrix with rows and columns labelled by the varieties, in which the entry in row A and column B is:

r if the varieties A and B are the same;

λ_{AB} if the varieties A and B are different.

The concurrence matrix of a balanced design with replication r and concurrence λ has diagonal entries r and non-diagonal entries λ.

Theorem 2.1

The concurrence matrix **C** of a ~~binary~~ incomplete block design is $\mathbf{BB}^T$, where **B** is the incidence matrix of the design.

If A is any variety, then

replication of A = number of 1s in row A of $\mathbf{B}$
= corresponding diagonal entry of $\mathbf{BB}^T$
= $\mathbf{C}$.

If A and B are different varieties, then

λ_{AB} = number of blocks containing both A and B
= number of 1s in the same position in rows A and B of $\mathbf{B}$
= corresponding non-diagonal entry of matrix $\mathbf{BB}^T$
= $\mathbf{C}$.

2 Balanced incomplete block designs

Balanced incomplete block design is frequently abbreviated to BIBD.

> **Theorem 2.2**
>
> Let Δ be a balanced incomplete block design with v varieties, replication r, block size k, and concurrence λ. Then
>
> $\lambda(v-1) = r(k-1)$.

The conditions

$k < v, \quad vr = bk, \quad r(k-1)/(v-1)$ is an integer,

are *necessary* for a balanced design: they are not *sufficient*.

> **Theorem 2.3: Fisher's inequality**
>
> In any balanced incomplete block design with v varieties and b blocks,
>
> $v \le b$.

A balanced incomplete block design is **symmetric** if $b = v$.

> **Theorem 2.4**
>
> For a symmetric BIBD in which v is even, the number $r - \lambda$ must be the square of an integer.

> **Theorem 2.5**
>
> Let Δ be a BIBD, and let $\overline{\Delta}$ be its complement. Then $\overline{\Delta}$ is also a BIBD.

> **Theorem 2.6**
>
> Let Δ be a BIBD, and let Δ^* be its dual. Then Δ^* is a BIBD if and only if Δ is symmetric.

3 Cyclic balanced designs

> **Theorem 2.7**
>
> The rows of the concurrence matrix of a cyclic design are obtained by taking the first row and shifting it one, two, three, ... places to the right, each time moving the right-hand number back to the beginning.

Arithmetic modulo n is performed on the numbers $\{0, 1, 2, \ldots, n-1\}$ by using ordinary arithmetic and then adding or subtracting an appropriate multiple of n to obtain a number in the set.

A **table of differences** is constructed by writing the starting block of a design on the left and at the top of the table; the entries in the table are obtained by subtracting each element at the top from each element on the left, using arithmetic modulo v, where v is the number of varieties.

> **Theorem 2.8**
>
> The first row of the concurrence matrix of a cyclic design is obtained by counting the number of times each variety occurs in the main body of the table of differences.

> **Theorem 2.9**
>
> A cyclic design is balanced if and only if the first block has the property that all non-zero varieties occur equally often among its differences.

A first block which gives rise to equal frequencies for the differences is called a *perfect difference set*.

A **perfect difference set** (modulo v) is a set of distinct numbers

$\{b_1, b_2, \ldots, b_k\}$ (modulo v)

such that the non-zero differences

$b_1 - b_2,\ b_1 - b_3, \ldots,\ b_k - b_{k-1}$

include each non-zero number (modulo v) equally often.

> **Theorem 2.10**
>
> If v is a prime number of the form $4n + 3$, where n is a non-negative integer, then the set of non-zero squares (modulo v) is a perfect difference set.

4 Constructions for BIBDs

Construction 1

Let p be a prime number of the form $4n + 3$.

To obtain a symmetric balanced design with parameters

$v = b = p, \quad r = k = (p-1)/2, \quad \lambda = (p-3)/4,$

take as first block the set of non-zero squares (modulo p); then construct the other blocks using the cyclic construction.

Construction 2

Let p be a prime number of the form $4n + 1$.

To obtain a balanced design with parameters

$v = p, \quad b = 2p, \quad r = p - 1, \quad k = (p-1)/2, \quad \lambda = (p-3)/2,$

take the set of non-zero squares (modulo p), and the set of non-squares (modulo p), and use the cyclic construction in each case; then put the two designs together.

Construction 3

Let p be a prime number of the form $4n + 1$.

To obtain a balanced design with parameters

$v = p + 1, \quad b = 2p, \quad r = p, \quad k = (p+1)/2, \quad \lambda = (p-1)/2,$

take the set of squares (modulo p) including 0, and the set of non-zero squares (modulo p) together with an extra variety z, and use the cyclic construction in each case; then put the two designs together.

Construction 4

Let p be a prime number of the form $4n + 3$.

To obtain a balanced design with parameters

$v = p + 1, \quad b = 2p, \quad r = p, \quad k = (p+1)/2, \quad \lambda = (p-1)/2,$

take the set of non-zero squares (modulo p) together with an extra variety z, and the set of non-squares (modulo p) together with 0, and use the cyclic construction in each case; then put the two designs together.

Section 3 · Special types of balanced design

1 Steiner triple systems

A **Steiner triple system** is a BIBD with $k = 3$ and $\lambda = 1$.

In any Steiner triple system:

- $r = (v-1)/2$;
- v is odd;
- $b = v(v-1)/6$.

> **Theorem 3.1**
>
> In any Steiner triple system, v has the form $6n + 1$ or $6n + 3$ for some integer n.

2 Finite projective planes

A **finite projective plane** is a regular incidence structure in which:

(a) for each pair of distinct points, there is exactly one line incident with both;

(b) for each pair of distinct lines, there is exactly one point incident with both.

A **finite projective plane of order n** is a symmetric BIBD with the parameters:

$$v = b = n^2 + n + 1, \quad r = k = n + 1, \quad \lambda = 1.$$

Theorem 3.2

The above two definitions of a projective plane are equivalent.

Theorem 3.3

If n is a power of a prime number, then there exists a finite projective plane of order n.

3 Latin square designs

A **latin square of side n** is an $n \times n$ square with n symbols arranged in such a way that each symbol occurs just once in each row and just once in each column.

4 Constructions for latin squares

Cyclic construction

In the first row, write down n letters in any order.

In each subsequent row, shift the letters one place to the right, moving the last one to the beginning.

Extended cyclic construction

In the first row, write down n letters in any order.

In each subsequent row, shift the letters r places to the right, moving the last r letters to the beginning; r can be any number from 1 to n such that r and n have no common factor.

Steiner triple system construction

Take a Steiner triple system with n varieties labelled $0, 1, \ldots, n-1$; label the rows and columns of the latin square similarly, and write $0, 1, \ldots, n-1$ down the main diagonal.

For each pair of varieties A and B, find the block of the Steiner triple system containing the varieties A and B, and place the third variety of that block in row A and column B of the latin square.

5 Uses of latin squares of side n

1 Two systems each containing n complete blocks, each block of one system having just one plot in common with each block of the other system, and n varieties.

2 One system containing n blocks of size n, and n^2 varieties, which are all the possible combinations of two treatment factors, each of which has n different 'levels'.

Section 4 Resolvable designs

1 Resolvability

A **replicate** is a set of plots containing each variety exactly once.

An incomplete block design is **resolvable** if its blocks can be grouped into replicates.

Theorem 4.1

In a resolvable incomplete block design, in which the number of blocks in each replicate is s

(a) $v = sk$; (b) $b = rs$.

Theorem 4.2

In any resolvable BIBD, $b \geq v + r - 1$.

2 Constructions for resolvable designs

A **(0, 1)-design** is a block design in which each concurrence is 0 or 1.

Construction of simple lattice design

Write down the k^2 varieties in any order in a $k \times k$ square array.

For the first replicate, take the *rows* of the square as blocks of the design.

For the second replicate, take the *columns* of the square as blocks of the design.

Construction of triple lattice design

Construct a simple lattice design with two replicates as above.

Construct a third replicate from the square array as follows.

The first block contains all varieties on the main (top-left to bottom-right) diagonal. The other blocks are the other 'diagonals' in the same direction.

Construction of rectangular lattice design

Write down the $k(k+1)$ varieties in any order in a $(k+1) \times (k+1)$ square array with the main diagonal missing.

For the first replicate, take the *rows* of the square as blocks of the design.

For the second replicate, take the *columns* of the square as blocks of the design.

3 Orthogonal latin squares and resolvability

Two $n \times n$ latin squares are **orthogonal** if, when they are superimposed, each of the n^2 possible ordered pairs of letters occurs just once.

Two orthogonal latin squares can be used to obtain replicates for the square lattice design.

To construct a resolvable design with n^2 varieties (numbers) write down an $n \times n$ square array, and two orthogonal $n \times n$ latin squares (using letters).

For the first replicate, take the blocks to be the *rows* of the square array.

For the second replicate, take the blocks to be the *columns* of the square array.

For the third replicate, associate a letter with each block and take the block to be the numbers in the square array corresponding to the positions of the chosen letter in the first latin square.

For the fourth replicate, proceed as for the third, using the second latin square.

Section 5 Balanced designs and codes

Theorem 5.1

Let Δ be a BIBD with v varieties, b blocks, replication r, block-size k, and concurrence λ. Let C be the code whose codewords are the rows of the incidence matrix. Then the Hamming distance between any two codewords is $2(r - \lambda)$, and the code C detects up to $r - \lambda$ errors and corrects up to $r - \lambda - 1$ errors.

Index

Printed in the United Kingdom.